Storage, Querying and Inferencing for Semantic Web Languages

Broekstra, J.

2005

SIKS Dissertation Series No. 2005-09

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

Promotiecommissie:
prof.dr. F. A. H. van Harmelen (promotor)
prof.dr. ir. G. J. Houben (Vrije Universiteit Brussel)
prof.dr. W. Nejdl (Universität Hannover)
prof.dr. D. Plexousakis (University of Crete, Heraklion)
prof.dr. A. Th. Schreiber (Vrije Universiteit Amsterdam)
prof.dr. J. Treur (Vrije Universiteit Amsterdam)

Cover illustration 'Timepiece' by Aaron Stainthorpe

ISBN 90-9019236-0

VRIJE UNIVERSITEIT

Storage, Querying and Inferencing for Semantic Web Languages

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op maandag 4 juli 2005 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Jeen Broekstra

geboren te Alkmaar

promotor: prof.dr. F.A.H. van Harmelen

dedicated to Jos van der Meer

Contents

Preface

Assiduus usus uni rei deditus et ingenium et artem saepe vincit.
– Cicero

PhD theses, although presented as the work of one person, are almost always the result of close cooperation between a group of people. This particular thesis is no exception to that rule; in fact, I feel very strongly that in my situation the work is so much a group effort that I am somewhat ashamed to be publishing this work with only my name on the cover. To make up for this, I will use this space to thank a number of people who have helped make this thesis happen.

In 1999, after receiving my MSc degree in Artificial Intelligence, my mind was not quite made up on what to do with my future: on the one hand, academic research seemed like an interesting career option, but on the other hand, I had spent the previous months as part of a team of software developers at Aduna (then still named Aidministrator), and I had felt right at home there. We developed software for navigation, analysis and visualisation of semi-structured information sources, and everything we were doing was new and fun and interesting.

While still in doubt about my future, the On-To-Knowledge project loomed on the horizon. Aduna was to be a major contributing partner in this project, and Jos van der Meer, founder and CEO of Aduna, came up with an idea: I could join the Aduna team as a full developer, and at the same time do a PhD. The work packages that Aduna needed to contribute to in the On-To-Knowledge project required innovative solutions, and such solutions require research. The close ties that Aduna already had with the Vrije Universiteit made the setup of this construction quite easy, and so I was suddenly both a software developer and a PhD student.

My inclusion in On-To-Knowledge was extremely lucky for me, as I got to contribute to some major developments that formed in large part the basis of what is now called the Semantic Web: the ontology language OIL, and its RDF Schema syntax. Together with Michel Klein, another PhD student at the Vrije Universiteit, I worked on grounding OIL firmly in Web standards. Brighter minds ran with the ideas developed there and the eventual result was the Web Ontology Language OWL. On-To-Knowledge also brought about the birth of Sesame, so it can well be argued that my thesis is, to a large extent, the final On-To-Knowledge deliverable.

The fact that I was not doing my PhD full time has led to a longer overall time for me to finish the project: five and a half years. It's been a great ride though, and I've had the

pleasure of meeting and working together with some extraordinary people.

First and foremost, the Aduna team have been a constant source of inspiration and support, and as said, I feel very strongly that this thesis is as much their work as it is mine. I therefore very sincerely wish to thank Arjohn Kampman, with whom I've worked most closely together as codeveloper of Sesame, Christiaan Fluit, Herko ter Horst, Jeroen Wester and Hilde Bleeker. Jos van der Meer, who is sadly no longer with us, also deserves my sincerest gratitude: without him this thesis would not have been possible.

I also wish to thank some equally extraordinary people at the Vrije Universiteit. First and foremost, my promotor, Frank van Harmelen, has been a constant support for me. I am by nature a pessimist and a doubter, but a conversation with Frank usually worked wonders in seeing the positive side of any development. Furthermore I wish to thank everyone I've worked or had lunch with in my time at the VU, especially Michel Klein, Heiner Stuckenschmidt and Marta Sabou, all of whom I've worked together with very closely in various stages of my PhD research.

This thesis has been written while listening to the music of My Dying Bride, so my thanks also extend to Aaron, Andrew, Hamish, Ade, Shaun and Sarah for making beautiful music and being a generally great bunch of people. Extra thanks go to Aaron for letting me use some of his art on the front cover.

Lastly, I wish to thank my family and friends. My parents and my brother deserve special mention for their unwavering confidence in the past years that I will succeed. Their confidence and support has helped me more than they probably realize.

Finally, I wish to thank Karen for coming into my life and for giving me the energy needed to finish this job.

Jeen Broekstra, Amersfoort, February 2005.

Chapter 1

Introduction

The term *Semantic Web* was coined by Tim Berners-Lee [Berners-Lee, 1998b] to capture the vision of a World Wide Web where information is shared, not just between human end users, but between machines as well. In other words, the Semantic Web is about enriching the current Web with machine processable data, to enable machines to share information and thus better help humans navigate, combine and retrieve information from the vast repository of knowledge that is today's Web.

In this work, we research the notion of Semantic Web from an implementational and representational perspective, and provide groundwork for the technical realization of Semantic Web tools and applications.

1.1 Ontologies and the Semantic Web

Currently, computers are changing from single isolated devices to entry points into a worldwide network of information exchange and business transactions (cf. [Fensel, 2000]). Support in data, information, and knowledge exchange is becoming the key issue in current computer technology. *Ontologies* will play a major role in supporting information exchange processes in various areas. The notion of ontology has become widespread in fields such as intelligent information integration, cooperative information systems, information retrieval, electronic commerce, and knowledge management. The reason ontologies are becoming so popular is in large part due to what they promise: a shared and common understanding of some domain that can be communicated between people and application systems. Because ontologies aim at consensual domain knowledge, their development is often a cooperative process involving different people, possibly at different locations.

Many definitions of ontologies have been formulated over the past decade. In our opinion, however, the one that best captures the essence of an ontology is based on the related definitions by [Gruber, 1993]: an ontology is a formal, explicit specification of a shared conceptualization. A *conceptualization* refers to an abstract model of some phenomenon in the world which identifies the relevant concepts of that phenomenon.

Explicit means that the type of concepts used and the constraints on their use are explicitly defined. *Formal* refers to the fact that the ontology should be machine processible, i.e. the machine should be able to interpret the information provided unambiguously. *Shared* reflects the idea that an ontology captures consensual knowledge, that is, it is not restricted to some individual, but accepted by a group.

The Semantic Web requires a generic mechanism for expressing machine readable semantics of data. The Resource Description Framework (RDF) [Beckett, 2004, Klyne and Carroll, 2004] is this foundation for processing metadata, providing a simple data model and a standardized syntax for metadata. It provides the language for writing down factual statements. On top of this, the next layer provides basic vocabulary and formal semantics for modeling ontologies. RDF Schema [Brickley and Guha, 2004, Hayes, 2004] provides such basic modeling primitives, allowing very simple ontologies to be formulated and shared. However, its expressivity is often not enough for full-fledged ontological modeling, so an extension in the form of additional primitives and accompanying formal semantics is required. OIL [Fensel et al., 2000a, Broekstra et al., 2001] and its successors DAML+OIL [Horrocks et al., 2001] and OWL [Dean and Schreijber, 2004] are such extended ontology languages.

The research presented in this thesis is concerned with the architecture of the Semantic Web, and the languages and tools needed to realize this vision.

1.2 Research Questions

The goal of this work is twofold:

1. to contribute to languages for representing and querying machine processable information, which enable sharing of such information.

2. to develop an architecture for storing, querying and reasoning with such machine processable information.

In order to realize these goals, we formulate the following research questions:

A: **How do we represent machine processable information on the Web?**
We need to be able to write down information in a way that enables systems to process them. Furthermore, the ground framework should be general enough to encompass a wide variety of applications, to enable the applicability of the approach accross the full spectrum of the Semantic Web.

B: **How do we access machine processable information on the Web?**
Once represented, we need ways to access and manipulate the machine processable information. This requires a conceptual access framework (e.g. a *query language*) and a technical architecture that implements such a conceptual framework.

C: **How do we create tools that manipulate machine processable information on the Web?**
When the conceptual framework for specifying and accessing machine processable

information is there, implementational frameworks for storing, querying, and reasoning with this information are needed.

1.3 Contributions of this Thesis

The main contribution of this thesis is the development of a technical framework for storage and querying of, and inferencing with, information encoded in a Semantic Web language. Several more specific contributions can be identified:

1. **Web-Based Knowledge Representation by Extending Existing Web Formalisms**
   We address research question A by proposing a way of 'layering' a formal knowledge representation language on top of existing Web languages, in order to enable the use of such languages in a heterogeneous environment.

2. **A Query Language for RDF**
   The Resource Description Framework (RDF) is the common foundational layer for representing machine processable information on the Web. We address research question B by specifying and implementing a query language for RDF, called SeRQL.

3. **Access APIs for manipulating and storing RDF**
   Programmatic support for Semantic Web languages is key for tool development in this area. We address research question C by developing a framework that enables developers to efficiently store, query and manipulate information that is encapsulated in RDF.

4. **Inferencing Strategies for RDF**
   The RDF specifications include a formal semantics [Hayes, 2004] and a proof system that allows simple entailments. We again address research question C, by providing algorithms and implementations of reasoners that can efficiently deal with this this proof system in real-world settings.

We will present these contributions successively in the following chapters. In the next section, we will give a detailed outline of the thesis that clarifies the relation between the chosen structure and the goals and contributions mentioned above.

1.4 Outline of this Thesis

This thesis is organized in two main parts, each covering a specific aspect of the overall problem of handling machine processable information on the Web.

- **Part I: Representation and Query Languages for the Semantic Web**
  In the first part of the thesis we will discuss several languages for representing machine processable information, and for accessing such information, once encapsulated.

- **Part II: Implementing Middleware for the Semantic Web**
  The second part of this thesis deals with creating a development framework that enables tool developers to use the languages described in part I. We discuss Programmatic APIs, query language implementations, storage facilities and scalability issues. Furthermore we discuss algorithms for dealing with inferencing, and the related performance issues.

The work is organized in chapters as follows:

Part I

Chapter 2 addresses research question A by introducing several Semantic Web languages, such as XML and RDF, and showing how an existing knowledge representation language, such as OIL, can be layered on top of these Web formalisms. We also briefly discuss the relation between OIL and the Web Ontology Language OWL.

Chapter 3 prepares for research question B by surveying several existing proposals for query languages and identifying their strengths and weaknesses.

Chapter 4 directly addresses research question B by presenting a new query language, SeRQL, and showing how this language enables expressive access patterns to RDF information. We also briefly describe future extensions of the SeRQL language.

Part II

Chapter 5 addresses research question C by introducing a Java development framework for storing, querying and inferencing for RDF, called Sesame. We describe the architecture and design choices of the system, and study issues of scalability and performance.

Chapter 6 addresses a specific subpart of research question C, by looking at inferencing for RDF. We present several algorithms and test implementations in the Sesame framework. We address issues of complexity, scalability and tradeoffs between reasoning time and storage space.

Chapter 7 further addresses research question C by investigating a side effect of a forward chaining inferencing strategy, namely the problem of truth maintenance, or keeping stored RDF models consistent when removing information. We present several algorithms for dealing with this problem and discuss the costs and benefits of each approach.

Chapter 8 wraps up this thesis by illustrating the languages, tools and strategies presented in the previous chapters in a real-world case study. The use of the Sesame development framework in a distributed, heterogeneous environment is shown, thus addressing this aspect of research question C. We first discuss the general notion of distributed querying and how it applies to the Sesame framework. We proceed to discuss

a case study in distributed querying using Sesame.

Chapter 9 summarizes the results of this thesis. We collect and explain the most important insights that the various chapters have given us in the nature of the problem of handlinge large amounts of heterogenous data in a Web environment. We review the work in terms of our research goals and briefly point to future work that may extend and improve the results presented.

Part I

Representation and Query Languages for the Semantic Web

Chapter 2

Representation Languages for the Semantic Web

The current approach to the Semantic Web is based on a stack of languages and protocols that are specifically designed to capture and communicate domain knowledge to diverse entities, without need for user interference. In other words, these languages aim to provide *machine processable* semantics of domain knowledge. In this chapter, we will look at several languages that lay at the core of the Semantic Web. First, will introduce the languages RDF and RDF Schema. Then we will introduce the ontology language OIL and explain how we designed the OIL serialization as an extension of RDF Schema. We will discuss several implications of this approach. We will also give a brief historical overview of the development of these languages, from the original RDF working drafts to the current revised specification, and how the development of OIL (and later DAML+OIL and OWL) have played a significant part in that development. The work presented in this chapter has originally been published in [Broekstra et al., 2001].

2.1 Introduction

OIL (Ontology Inference Layer), a major spin-off of the IST project On-To-Knowledge[1], is a Web-based representation and inference layer for ontologies, which unifies three important aspects provided by different communities: formal semantics and efficient reasoning support as provided by Description Logics, epistemological rich modeling primitives as provided by the Frame community, and a standard proposal for syntactical exchange notations as provided by the Web community.

In this chapter, we will show how RDF Schema (RDFS) can be extended to contain a more expressive knowledge representation language, which would enrich it with

[1] *On-To-Knowledge: Content-driven Knowledge-Management Tools through Evolving Ontologies* (IST-1999-10132). Project partners are the Vrije Universiteit Amsterdam (VU); the Institute AIFB, University of Karlsruhe, Germany; AIdministrator, the Netherlands; British Telecom Laboratories, UK; Swiss Life, Switzerland; CognIT, Norway; and Enersearch, Sweden. `http://www.ontoknowledge.org/` [Fensel et al., 2000b]

the required additional expressivity and the semantics of that language. We will do this by describing the ontology language OIL as an extension of RDFS. Enabling the use of more expressive formal languages on the Web offers the basis for inferences, and thus for automated services, such as information filtering and query answering. An important advantage peculiar to our approach is that our extension method ensures maximal sharing of meta-data on the Web: even partial interpretation of an OIL ontology by less semantically aware processors will yield a correct partial interpretation of the meta-data.

This chapter is structured as follows. In section 2.2 we briefly introduce XML and XML Schema. Section 2.3 presents an introduction to RDF and RDF Schema. Section 2.4 contains an introduction into OIL. Section 2.5 illustrates in detail how RDF Schema can be extended, using OIL as an example of a knowledge representation language. The result is an RDF Schema definition of OIL primitives, which makes it possible to express any OIL ontology in RDF syntax. In section 2.6 we discuss how our approach makes it possible to tap into the additional advantages of OIL on the Web, such as reasoning support and formal semantics, while retaining maximal compatibility with 'pure' RDF(S).

2.2 XML

The Extensible Markup Language (XML) [Bray et al., 1998] describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO8879:1986, 1986]. By construction, XML documents are conforming SGML documents. XML, while named a markup language, is actually a markup meta language. It allows one to define a set of markup tags, which can be chosen to reflect the domain specific semantics of the information, rather than merely its layout and structure (as is the case in, for example, HTML). An XML document consists of a properly nested set of open and close tags, where each tag can have a number of attribute-value pairs. The vocabulary of the tags and their allowed combination is not fixed, but can be defined per application of XML. An example XML document is:

```
<?xml version="1.0"?>
<body>
  This page is written by
  <author>Frank van Harmelen</author>.
  <location>
    His tel.nr. is
    <tel>47731</tel>,
    and his room number is
    <room>T3.57</room>.
  </location>
</body>
```

From the indentation of the above example, it is easy to see that the basic data model of XML is a labeled tree, where each tag corrsponds to a labeled node in the data model, and each nested subtag is child in the tree. The intended structure of XML documents within a particular domain, i.e. the allowed labels and the way in which they can be nested, is described in a Document Type Declaration (DTD), which expresses in a grammar-like formalism which allowed sequences and nestings of tag are allowed. For example, a DTD for the above XML file would possibly look like this:

```
<?xml version="1.0"?>
<!ELEMENT body (author, location, #PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT location (tel*, room?, institution?, city?, #PCDATA)>
<!ELEMENT tel (#PCDATA)>
<!ELEMENT room (#PCDATA)>
<!ELEMENT institution (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

DTDs offer very limited expressivity. For example, it is only possible to describe the legal lexical nesting of elements, no other relationships between elements can be expressed. Also, virtually no typing of values is possible. These shortcomings make DTDs unsuited for real schema modeling (in fact, it was always intended as a 'stop-gap' method until a more suitable schema language was developed). Developments are well underway at the W3C to replace DTDs with XML Schema definitions [Thompson et al., 2001, Biron and Malhotra, 2001].

In practice, XML is being used for a number of rather different purposes:

- as a serialization syntax for other markup languages. For example, the SMIL multimedia markup language [Hoschka, 1998] uses the XML format for its markup syntax. Syntactically, SMIL is simply a particular XML DTD. The real value of SMIL lies in the fact that there exists a common understanding of the intended meaning of the elements of that particular DTD. However, it is important to realize that the DTD only specifies the syntactic conventions, and that any such intended semantics remain outside the realm of the XML specification.

- as semantic markup of Web-pages (as seen in the sample XML document above).

- as a uniform data-exchange format. The above example can also be a data object transferred between two applications. Again, only the syntactic structure of XML is enforced; the intended meaning of the various elements is entirely implicit in the XML document.

Because the intended semantics of XML tagging is always implicit (since neither the XML definition itself nor the DTD specify anything but syntax), only the first usage of XML is in real accordance with the original goal of the language. It is especially important that due to the fact that XML technology focuses on structure and syntax, and has no regard for semantics, it is unsuited to be used as an Ontology language, without an additional protocol layer on top of it that defines notions of semantics. In section 2.3, we will see how RDF and RDF Schema fulfill this role of semantic layer on top of XML.

2.2.1 XML Schema

XML Schemas are a means for defining constraints on valid XML documents. The XML Schema Specification is divided into two parts: in [Thompson et al., 2001] Structures are described, and in [Biron and Malhotra, 2001] Datatypes are described. A human readable explanation on XML Schemas can be found in [Fallside and Walsmley, 2004].

XML Schemas have the same purpose as DTDs, but they provide several significant improvements:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/1999/XMLSchema">
  <complexType name="address">
    <element name ="name" minOccur="1" maxOccur="1" type="string"/>
    <element name="street" minOccur="1" maxOccur="2" type="string"/>
    <element name="city" minOccur="1" maxOccur="1" type="string"/>
    <element ref="zip" minOccur="1" maxOccur="1"/>
  </complexType>
  <element name="zip" type="zipCode"/>
  <simpleType name="zipCode" base="string">
    <pattern value="[0-9]5(-[0-9]4)?"/>
  </simpleType>
</schema>
```

Figure 2.1: an example XML schema

- XML Schema definitions are XML documents themselves. For example in figure 2.1 the schema definition for an 'address' tag is given. The schema itself is in XML, whereas a traditional DTD would provide such a definition in an external other language. The advantage is that all tools developed for XML (e.g. validation and rendering tools) can immediately be applied to Schemas as well.

- XML Schema provides a rich set of datatypes that can be used to define the values of elementary tags.

- XML Schema provides much richer means for defining nested tags (i.e., tags with subtags).

- XML Schema provides the namespace mechanism to combine XML documents with heterogeneous vocabulary.

- XML Schema provides a mechanism to define new datatypes, by extending or constraining existing datatypes to form new subtypes. For example in figure 2.1 we see a simple datatype zipCode which is defined by constraining the basic datatype string to allow only certain patterns as values.

A more detailed discussion of XML Schema can be found in [Klein et al., 2000]. At first sight XML Schema may seem an attractive language for ontology definitions with datatyping and type extension mechanisms. However, the conclusion of [Klein et al., 2000] is that, due to its focus on structure rather than domain knowledge content, XML Schema is unsuited for this purpose and should only be used for describing the syntactic structure of documents.

2.3 RDF and RDF Schema

This section presents the main features of RDF and RDF Schema.

2.3.1 Introduction to RDF

The Semantic Web requires machine-processible semantics in information. The Resource Description Framwork (RDF) [Beckett, 2004] is a foundation for processing metadata; it provides interoperability between applications that exchange machine-processible information on the Web. Basically, RDF defines a data model for describing machine-processible semantics in data. The basic data model consists of three object types:

- **Resources**: A resource may be an entire Web page; a part of a Web page; a whole collection of pages; or an object that is not directly accessible via the Web; e.g. a printed book. Resources are always named by URIs.
- **Properties**: A property is a specific aspect, characteristic, attribute, or relation used to describe a resource.
- **Statements**: An RDF statement consists of a specific resource, together with a named property and the value of that property for the resource in question.

These three individual parts of a statement are called the *subject*, *predicate*, and the *object*, respectively. In a nutshell, RDF defines object-property-value-triples as basic modeling primitives and introduces a standard syntax for them. An RDF document will define properties in terms of the resources to which they apply. For example:

```
<rdf:RDF>
  <rdf:Description rdf:about="http://www.w3.org">
     <Publisher>World Wide Web Consortium</Publisher>
  </rdf:Description>
</rdf:RDF>
```

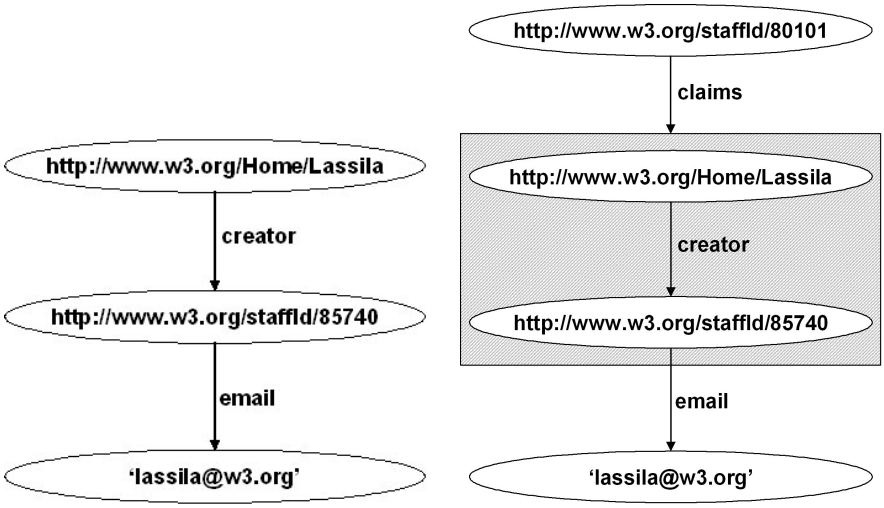
states that `http://www.w3.org` (the subject) has as its publisher (the predicate) the W3C (the object). Since both the subject and the object of a statement can be resources, these statements can be linked in a chain:

```
<rdf:RDF>
  <rdf:Description rdf:about="http://www.w3.org/Home/Lassila">
    <Creator rdf:resource="http://www.w3.org/staffId/85740"/>
  </rdf:Description>

  <rdf:Description rdf:about="http://www.w3.org/staffId/85740">
    <Email>lassila@w3.org</v:Email>
  </rdf:Description>
</rdf:RDF>
```

States that `http://www.w3.org/Home/Lassila` (the subject) was created by staff member 85740 (the object). In the next statement, this same resource (staff member 85740) acts as the subject, who states that his email address is `lassila@w3.org`. In figure 2.2, the resulting graph is represented.

The object of a statement can be either a resource or a *literal*, that is, a string value (for example the e-mail address in the above example is represented as a literal). Such literals can optionally have an XML datatype [Biron and Malhotra, 2001] or a language identifier attached.



(a) Two Chained Statements (b) Reified Statement

Figure 2.2: Example RDF graphs

Finally, RDF introduces a form of *reification*, in which statements are themselves also resources. Thus, statements can be applied recursively to statements, allowing their nesting. For example, in figure 2.2 we see how reification is used to make the statement that staff member 80101 claims that the creator of the webpage `http://www.w3.org/Home/Lassila` is staff member 85740.

Thus, the underlying data model of RDF is a directed labeled hyper-graph, and each statement acts as a predicate-labeled link between object and subject. The graph is a hyper-graph since each node, in itself, can again contain an entire graph.

Since the XML syntax of RDF is rather verbose, we will, in this thesis, often use a more compact notation for RDF statements, as follows: `(subject predicate object)`, for example:
`(http://www.w3.org/Home/Lassila Creator http://www.w3.org/staffId/85740).`

2.3.2 Introduction to RDF Schema

The modeling primitives offered by RDF are very basic[2]. Therefore, the RDF Schema specification [Brickley and Guha, 2004] defines further modeling primitives in RDF. In other words, RDF Schema extends (or enriches) RDF by assigning an externally specified semantics to specific resources, e.g., to `rdfs:subClassOf`, to `rdfs:Class` etc. It is only because of these external semantics that RDF Schema is useful. Moreover, these semantics cannot be captured in RDF. (If that were possible, there would be no

[2] Actually they correspond to binary predicates of ground terms, where, however, the predicates may be used as terms, as well.

need for RDFS). OIL bears a similar relationship to RDFS: by defining semantics for specific resources we can further extend (or enrich) RDF Schema. This allows OIL to capture meaning that cannot be captured in RDFS; and this is where its added value lies. Furthermore, we will be careful to create this extension to RDF Schema in such a way that a partial interpretation without the additional OIL semantics will still yield a valid RDF Schema interpretation.

As a simple example of the use of RDF Schema, we introduce a class `W3CStaffMember` and a class `Person` to extend our RDF example with:

```
<rdf:RDF>
  <rdf:Description rdf:about="http://www.w3.org/ontology/Person">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:about="http://www.w3.org/ontology/W3CStaffMember">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/ontology/Person"/>
  </rdf:Description>
</rdf:RDF>
```

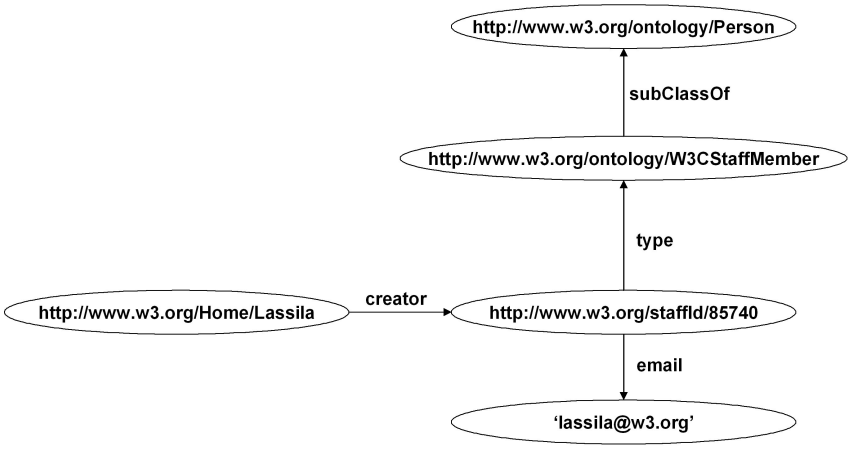


Figure 2.3: Example RDF Schema use

In figure 2.3, we see how these classes can be used to specify the types of certain instances in our RDF example: we explicitly specify that resource identified by the staff number 85740 is of type `W3CStaffMember`. It is worth noting that due to the semantics of RDF Schema, it is implicitly true (by inheritance) that it is also of type `Person`. It is also worth noting that even though we use RDF Schema notions, the data model is still a graph: RDF Schema only defines new vocabulary and a semantics for that vocabulary.

Despite the similarity in their names, RDF Schema fulfills a very different role than does XML Schema. RDF Schema merely defines additional vocabulary. XML Schema, like DTDs, prescribes the order and combination of tags in an XML document. In contrast, RDF Schema only provides information about the interpretation of the statements

in an RDF data model, but does not constrain the syntactical appearance of an RDF description. Therefore, the definition of OIL in RDFS presented in this document will not include constraints on the structure of an actual OIL ontology.

In this section we will briefly discuss the overall structure of RDFS and its main modeling primitives.

The data model of RDF Schema

Figure 2.4 presents the subclass-of hierarchy of RDFS and figure 2.5 presents the instance-of relationships of RDFS primitives according to [Brickley and Guha, 2004]. The 'rdf' prefix refers to the RDF name space (i.e., primitives with this prefix are already defined in RDF) and 'rdfs' refers to new primitives defined by RDFS. Note that RDFS uses a non-standard object-meta model: the properties `rdfs:subClassOf`, `rdf:type`, `rdfs:domain` and `rdfs:range` are used both as primitive constructs in the definition of the RDF Schema specification and as specific instances of RDF properties. This dual role makes it possible to view - say - `rdfs:subClassOf` as an RDF property just like other predefined or newly introduced RDF properties. However, it does introduce a self referentiality into the RDF Schema definition, which makes it rather unique as compared to conventional model and meta modeling approaches, and makes the RDF Schema specification very difficult to read and to formalize, cf. [Nejdl et al., 2000].

The modeling primitives of RDF Schema

In this section, we will discuss the main classes, properties, and constraints in RDFS. A formal specification of the semantics of these primitives can be found in [Hayes, 2004], and is also briefly discussed in chapter 6.

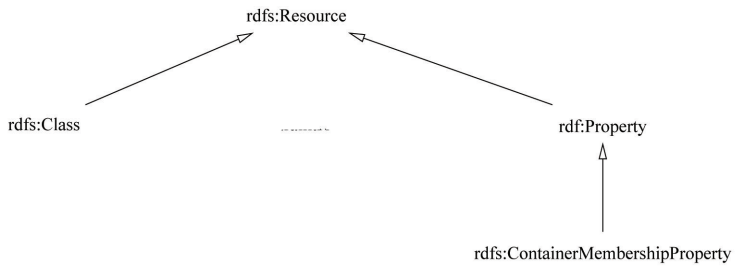


Figure 2.4: The subclass-of hierarchy of modeling primitives in RDFS.

- **Core classes** are `rdfs:Resource`, `rdf:Property`, and `rdfs:Class`. Everything that is described by RDF expressions is viewed to be an instance of the class `rdfs:Resource`. The class `rdf:Property` is the class of all properties used to characterize instances of `rdfs:Resource`, i.e., each slot / relation is an instance of `rdf:Property`. Finally, `rdfs:Class` is used to define concepts in RDFS, i.e., each concept must be an instance of rdfs:Class.

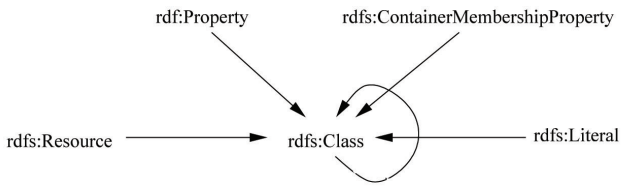


Figure 2.5: The instance-of relations of modeling primitives in RDFS.

- **Core properties** are `rdf:type`, `rdfs:subClassOf`, and `rdfs:subPropertyOf`. The `rdf:type` relation models instance-of relationships between resources and classes. A resource may be an instance of more than one class. The `rdfs:subClassOf` relation models the subsumption hierarchy between classes and is supposed to be transitive. Again, a class may be a subclass of several other classes. The `rdfs:subPropertyOf` relation models the subsumption hierarchy between properties. If some property P_2 is a rdfs:subPropertyOf another property P_1, and if a resource R has a P_2 property with a value V, this implies that the resource R also has a P_1 property with a value V.

- **Core constraints** are `rdfs:range` and `rdfs:domain`, which can be used to couple properties with value and subject classes in a global way. Multiple domain/range constraints on a single properties are interpreted through conjunctive semantics: if a property P has as its domain classes A and B, an instance a that is the subject of a statement using P, is entailed to be an instance of both A and B.

2.3.3 Changes in RDF and RDF Schema

The description of RDF and RDF Schema in the previous section reflects the current set of W3C Recommendations. This revised set of recommendations is the result of quite a few changes that have been made to the original specifications (see [Lassila and Swick, 1999, Brickley and Guha, 2000]) since OIL was originally developed, some of which are very relevant for this chapter (and in fact were changes made as a direct result of experiences gained from the design of OIL). In this section we will briefly highlight the relevant changes and discuss the reasons for changes where applicable.

First and foremost, the original RDF specification as defined in [Lassila and Swick, 1999, Brickley and Guha, 2000] had no formal semantics. The reason for its introduction was criticism on the part of several tool developers: it turned out that the RDF specification, by lack of a formal semantics, was imprecise, leading to incompatibilities between different implementations of the specification. For example, the RDF Suite [Alexaki et al., 2000] and Sesame [Broekstra et al., 2002] toolkits both implemented the same

query language (RQL [Karvounarakis et al., 2002]), yet because of differences in the interpretation of the RDF models queries gave different answers depending on the tool used to process them. In recognition of this shortcoming, the RDF Model Theory [Hayes, 2004] was developed.

Several specific things regarding RDF/RDFS primitives have changed as well. One of the most important changes is the definition of `rdfs:subClassOf` and `rdfs:subPropertyOf`. In the original specification, the subsumption hierarchies formed by these relations were required to be free of cycles. In [Broekstra et al., 2001] however, we argued that this restriction should be dropped: without cycles one cannot even represent equivalence between two classes – in our view this is an essential modeling primitive for any knowledge representation language. Moreover, these kinds of constraints significantly add to the complexity of parsing and validating RDF documents in a way which we think would be highly undesirable. When cycle detection is mandatory, parsing an RDF document in a streaming fashion is no longer feasible: a two-step parsing technique has to be used to detect that certain constructs refer to each other in a cyclic fashion. Moreover, such restrictions are really semantic constraints rather than syntactic ones (they limit the kinds of models that can be represented), even if the reasoning required in order to detect constraint violation is of a very basic kind.

Second, the original RDF specification (as specified in [Lassila and Swick, 1999, Brickley and Guha, 2000]) allowed only a single range restriction on a property. Although this can be circumvented by defining a dummy superclass of all classes in the range restriction, we see no reason for this restriction in RDFS. From a modeling point of view, allowing more than one range restriction is a much cleaner solution. Again, after presenting this argument in [Broekstra et al., 2001] and subsequent discussions with the RDF community[3], the restriction was dropped from the RDF specifications.

Third, the semantics of multiple domain restrictions were loosely defined as disjunctive (that is, if a property p has as its domain the classes A and B, the set of possible values for its subject is defined as the *union* of the two classes). In a mailinglist discussion with the RDF Interest Group [4] we argued that this should be changed to an intersection interpretation:

> (...) On the intended semantics of `rdfs:domain`, we believe that this should be changed to intersection semantics as well. (...) If you add a local domain restriction that says, for example, that the domain restriction on "ISBN-number" should be "book", then given the union semantics, this has no effect at all if elsewhere it has already been asserted that the domain restriction on "ISBN-number" is "document". If I assert $\texttt{rdfs:domain}(p, s)$ and I know that $p(y, x)$, then I should be able to assume $\texttt{rdf:type}(y, s)$ (...):
>
> $\texttt{rdf:type}(y, s) \leftarrow \texttt{rdfs:domain}(p, s) \wedge p(y, x)$
>
> With union semantics, this cannot be inferred. In fact, given that you can't know about all the other domain restrictions that have been made "elsewhere", then $\texttt{rdfs:domain}(p, s)$ becomes completely meaningless.

[3] See a.o. `http://lists.w3.org/Archives/Public/www-rdf-interest/2000Aug/0095.html`

[4] See `http://lists.w3.org/Archives/Public/www-rdf-interest/2000Sep/0132.html`

In later versions of the RDF specifications, the semantics of domain restrictions were adapted accordingly.

A last significant change in the RDF specifications was the introduction of datatypes. The original RDF specification had no such feature, but in later versions XML Schema datatypes were introduced as a datatyping mechanism for literals. However, when the OIL specification presented in the following sections was designed, this datatyping mechanism was not yet available. Hence, the OIL specification defines its own datatypes instead of reusing RDF datatypes.

In the following sections, we will present the knowledge representation language OIL, and its specification as an extension of RDF(S), as it was originally defined. Therefore it will occassionaly refer to this earlier version of RDF.

2.4 OIL

This section offers a brief history of and introduction to the OIL language; more details can be found in [Horrocks et al., 2000].

OIL was one of the first attempts to define a knowledge representation language explicitly for use on the (Semantic) Web. Its design goals came from the stated goals of the On-To-Knowledge project in which it was developed: to enable sharing and reusing of ontological knowledge in large corporate ennvironments. It was immediately realized that using existing Web standards as the basis would be of great benefit in realizing this vision.

Independently, the US DAML project[5] had defined a language called DAML-ONT [Stein et al., 2000], which had very similar goals. In recognition of the overlap of the two languages, the projects worked together to create a language called DAML+OIL [Horrocks et al., 2001], which essentially reworked the original DAML-ONT proposal to use more of the ideas first proposed in the OIL language. DAML+OIL in its turn has been the basis for the development of the official W3C Web Ontology Language, OWL [Dean and Schreijber, 2004]: its stated design goal was explicitly to take DAML+OIL as a starting point and make minimal changes only. OWL DL is the language dialect of OWL that most closely approach this design goal.

Due to the introduction of OWL, the OIL language has become obsolete. However, we still present it in this thesis for two reasons:

- the original work on OIL and its specification as an extension of RDF (to which the author of this thesis has heavily contributed) has formed the groundwork for what is now OWL;

- due to compromises made in the merge with the original DAML-ONT specification, the way in which OWL extends RDF Schema is not quite as clean (in terms of compatibility of semantics) as the original OIL specification. We will show a brief example of this in section 2.7.

[5] `http://www.daml.org/`

In the following, we introduce the OIL language and its design goals. A small example of an ontology in OIL is presented in figure 2.6.

This language has been designed such that:

1. it provides most of the modeling primitives commonly used in frame-based and Description Logic (DL) oriented Ontologies;

2. it features simple, clean and well-defined first-order semantics;

3. automated reasoning support, (e.g., class consistency and subsumption checking) can be provided. The FaCT system [Bechhofer et al., 1999], a DL reasoner developed at the University of Manchester, can be - and has been - used to this end [Stuckenschmidt, 2000].

An ontology in OIL is represented via an *ontology container* and an *ontology definition* segment. For the container, we adopt the components defined by Dublin Core Metadata Element Set, Version 1.1[6].

The ontology-definition segment consists of an optional import statement, an optional rule base and class, slot and axiom definitions.

A class definition (**class-def**) associates a class name with a class description. This class description, in turn, consists of the type of the definition (either primitive, which means that the stated conditions for class membership are necessary but not sufficient, or defined, which means that these conditions are both necessary and sufficient), a subclass-of statement and zero or more slot-constraints.

The value of a **subclass-of** statement is a (list of) class-expression(s). This can be either a class name, a slot-constraint, or a boolean combination of class expressions using the operators **and**, **or** and **not**, with the standard DL semantics.

In some situations it is possible to use a *concrete-type-expression* instead of a class expression. A concrete-type-expression defines a range over some data type. Two data types that are currently supported in OIL are **integer** and **string**. Ranges can be defined using the expressions (**min** X), (**max** X), (**greater-than** X), (**less-than** X), (**equal** X) and (**range** X Y). For example, (**min** 21) defines the data type consisting of all the integers greater than or equal to 21. As another example, (**equal** "xyz") defines the data-type consisting of the string "xyz".

A **slot-constraint** (or property restriction) is a list of one or more constraints (restrictions) applied to a slot (property). Typical constraints are:

- **has-value (class-expr)** Every instance of the class defined by the slot constraint must be related, via the slot relation, to an instance of each class expression in the list.

- **value-type (class-expr)** *If* an instance of the class defined by the slot-constraint is related via the slot relation to some individual x, *then* x must be an instance of each class-expression in the list.

[6] See `http://purl.org/DC/`

ontology-container
title "African Animals"
creator "Ian Horrocks"
subject "animal, food, vegetarians"
description "A didactic example ontology describing African animals and plants"
description.release "2.0"
publisher "I. Horrocks"
type "ontology"
format "pdf"
identifier "http://.../oil-rdfs.pdf"
source "http://www.africa.com/"
language "en-uk"

ontology-definitions
slot-def *eats*
inverse *is-eaten-by*

slot-def *has-part*
inverse *is-part-of*
properties transitive

slot-def *weight*
range (**min** 0)
properties functional

slot-def *colour*
range string
properties functional

class-def animal

class-def plant

disjoint animal plant
class-def tree
subclass-of plant

class-def branch
slot-constraint *is-part-of*
has-value tree

class-def leaf
slot-constraint *is-part-of*
has-value branch

class-def defined carnivore
subclass-of animal
slot-constraint *eats*
value-type animal

class-def defined herbivore
subclass-of animal
slot-constraint *eats*
value-type (plant **or**
(**slot-constraint** *is-part-of*
has-value plant))
disjoint carnivore herbivore
class-def mammal
subclass-of animal

class-def elephant
subclass-of herbivore mammmal
slot-constraint *eats*
value-type plant
slot-constraint *colour*
has-filler "grey"

class-def defined african-elephant
subclass-of elephant
slot-constraint *comes-from*
has-filler Africa

class-def defined indian-elephant
subclass-of elephant
slot-constraint *comes-from*
has-filler India

disjoint-covered elephant **by**
african-elephant indian-elephant
——— instance information ———

instance-of Africa continent
instance-of Asia continent
related *is-part-of* India Asia

Figure 2.6: An example OIL ontology, modeling the animal kingdom

- **max-cardinality n (class-expr)** An instance of the class defined by the slot-constraint can be related to at most n distinct instances of the class-expression via the slot relation (also min-cardinality and, as a shortcut for both min and max, cardinality).

A slot definition (**slot-def**) associates a slot name with a slot definition. A slot definition specifies global constraints that apply to the slot relation. A slot-def can consist of a **subslot-of** statement, **domain** and **range** restrictions, and additional qualities of the slot, such as **inverse** slot, transitive, and symmetric.

An *axiom* asserts some additional facts about the classes in the ontology, for example that the classes carnivore and herbivore are disjoint (that is, have no instances in common). Valid axioms are:

- **disjoint (class-expr)+** All of the class expressions in the list are pairwise disjoint.
- **covered (class-expr) by (class-expr)+** Every instance of the first class expression is also an instance of at least one of the class expressions in the list.
- **disjoint-covered (class-expr) by (class-expr)+** Every instance of the first class expression is also an instance of exactly one of the class expressions in the list.
- **equivalent (class-expr)+** All of the class expressions in the list are equivalent (i.esemweblanguages they have the same instances).

The syntax of OIL is oriented towards XML and RDF. [Horrocks et al., 2000] defines a DTD and a XML schema definition for OIL. [Klein et al., 2000] derives an XML Schema for writing down instances of an OIL ontology. Here, we will derive the RDFS syntax of OIL.

2.5 OIL as an extension of RDF Schema

RDF provides basic modeling primitives: ordered triples of objects and links. RDFS enriches this basic model by providing a vocabulary for RDF, which is assumed to have a certain semantics. In this section we will provide a careful analysis of the relation between RDFS and OIL by defining OIL in RDFS, using existing vocabulary as much as possible. The reason for this is twofold. First, by re-using RDFS primitives we are effectively imposing a formal semantics on them, specifically the formal semantics of OIL. Second, because we only extend RDFS with new primitives where necessary, RDFS becomes a full sub-language of OIL, thus providing backward compatibility from OIL to RDFS. It is worth noting that in OIL's successor language OWL, this relation is less clean due to a less careful way of extending RDFS (see section 2.7 for an example).

The complete schema can also be found at `http://www.ontoknowledge.org/oil/rdf-schema/`. The RDFS serialization of the example from the previous section is available at `http://www.ontoknowledge.org/oil/a-animals.rdfs`.

2.5.1 The ontology container and import mechanism

The outer box of the OIL specification in RDFS is defined by the XML prologue and the namespace definitions `xmlns:rdf` and `xmlns:rdfs`, which refer to RDF and RDFS, respectively. Namespace definitions make externally defined RDF constructs available for local use. Therefore, the OIL specification uses RDF and RDFS, and an actual ontology in OIL has namespace definitions which make both the RDF and RDFS definitions as well as the OIL specification itself available.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:oil="http://www.ontoknowledge.org/
               oil/rdf-schema/2000/11/10-oil-standard"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:dcq="http://purl.org/dc/qualifiers/1.1/"
    <!-- The ontology defined in OIL with RDFS syntax-->
</rdf:RDF>
```

It is important to notice that namespace definitions are not import statements, and are therefore not transitive. An actual ontology also has to define the namespaces for RDF and RDFS via `xmlns:rdf` and `xmlns:rdfs`, otherwise, all elements of OIL that directly correspond to RDF and RDFS elements would not be available.

The **ontology-container** of OIL provides metadata describing an OIL ontology. Because the structure and RDF-format of the Dublin Core element set is used, it is enough to import the namespace of the Dublin Core element set. Note that the fact that an OIL ontology should provide a container definition is an *informal* guideline in its RDFS syntax, because it is not possible to enforce this in the schema definition.

Apart from the container, an OIL ontology consists of a set of definitions. The **import** definition is a simple list of references to other OIL modules that are to be included in this ontology. We make use of the XML namespace mechanism to incorporate this mechanism in our RDFS specification. Notice again that, in contrast to the import statement in OIL, 'inclusion' via the namespace definition is not transitive.

2.5.2 Class and attribute definitions

In OIL, a class definition links a class with a name, a documentation, a type, its superclasses, and the attributes defined for it. In RDFS, classes are simply declared by giving them a name (with the ID attribute). We will show how OIL class definitions can be written down in RDF, while trying to make use of existing RDFS constructs as much as possible, but where necessary extending RDFS with additional constructs (see table 2.1 and figure 2.7).We conform to the informal RDF guideline to start property names with a lower-case letter, and class names with a capital.

To illustrate the use of these extensions, we will walk through them by means of some example OIL class definitions that need to be represented in RDFS syntax:

class-def defined herbivore
  subclass-of animal
    slot-constraint *eats*

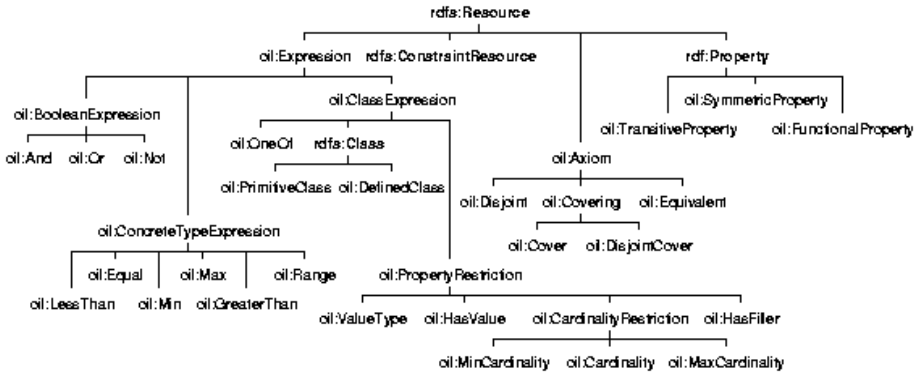


Figure 2.7: The OIL extensions to RDFS in the subsumption hierarchy.

value-type (plant **or**
(**slot-constraint** *is-part-of* **has-value** plant))

class-def elephant
subclass-of herbivore mammal
slot-constraint *eats*
value-type plant
slot-constraint *colour*
has-filler "grey"

The first defines a class "herbivore", a subclass of animal, whose instances eat plants or parts of plants. The second defines a class "elephant", which is a subclass of both herbivore and mammal.

Defined classes and Primitive classes

We start by translating the first class definition. The header can be done in a straightforward manner, using the rdfs:Class construct and the rdf:ID property to assign a name:

```
<rdfs:Class rdf:ID="herbivore"> </rdfs:Class>
```

From this definition it is not yet clear that this class is a defined class. We chose to introduce two extra classes in the OIL namespace, named PrimitiveClass and DefinedClass. In a particular class definition, we can use one of these two ways to express that a class is a defined class:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
</rdfs:Class>
```

or:

```
<oil:DefinedClass rdf:ID="herbivore"> </oil:DefinedClass>
```

We will use the first method of serialization throughout this article, but it is important to realize that both model exactly the same.

This way of making an actual class an instance of either DefinedClass or PrimitiveClass introduces a nice object-meta distinction between the OIL RDFS schema and the actual ontology: using rdf:type you can consider the class "herbivore" to be an *instance* of DefinedClass. In OIL in general, if it is not explicitly stated that a class is defined, the class is assumed to be primitive.

Class Subsumption

Next, we have to translate the subclass-of statement to RDFS. This also can be done in a straightforward manner, simply re-using existing RDFS expressiveness:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
</rdfs:Class>
```

However, if one wants to define a class as a subclass of a class *expression*, one should use the oil:subClassOf property.

Slot Constraints

We still need to serialize the slot constraint on the class "herbivore". In RDFS, there is no mechanism for restricting the attributes of a class on a local level. This is due to the property-centric nature of the RDF data model: properties are defined globally, with their domain description coupling them to the relevant classes.

To overcome this problem, we introduce the oil:hasPropertyRestriction property, which is an rdf:type of rdfs:ConstraintProperty (analogous to rdfs:domain and rdfs:range). Here we take full advantage of the intended extensibility of RDFS. We also introduce oil:PropertyRestriction as a placeholder class[7] for specific classes of slot constraints, such as has-value, value-type, cardinality and so on. These are all modeled in the OIL namespace as subclasses of oil:PropertyRestriction:

```
<rdfs:Class rdf:ID="ValueType">
  <rdfs:subClassOf rdf:resource="#PropertyRestriction"/>
</rdfs:Class>
```

and similar for the other slot constraints. For the three cardinality constraints, an extra property "number" is introduced, which is used to assign a concrete value to the cardinality constraints.

To connect a ValueType slot constraint with its actual values, such as the property it refers to and the class it restricts that property to, we introduce a pair of helper properties.

[7] A placeholder class in the OIL RDFS specification is only used to apply domain- and range restrictions to a group of classes, and will not be used in the actual OIL ontology.

These helper properties have no direct counterpart in terms of OIL primitives, but they serve to connect two classes. We define a property oil:onProperty to connect a property restriction with the subject property, and a property oil:toClass to connect the property restriction to the its class restriction.

In our example ontology, the first part of the slot constraint would be serialized using the primitives introduced above as follows:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass> </oil:toClass>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

If we would want to restrict the value type of a property to a string or a an integer, we could use the toConcreteType property:

```
...
  <oil:ValueType>
    <oil:onProperty rdf:resource="#age"/>
    <oil:toConcreteType
        rdf:resource="http://www.ontoknowledge.org/oil/
          rdf-schema/2000/11/10-oil-standard#Integer"/>
  </oil:ValueType>
...
```

Boolean Expressions

The slot constraint has not been completely translated yet: the toClass element is not yet filled. Here we come across a feature of OIL that is not available in RDFS: the *boolean expression*. A boolean expression is an expression that evaluates to either a class definition or a concrete type. In the case of a class definition, such an expression is a boolean combination of classes and/or slot constraints. In the case of a concrete type definition, the expression can be a simple string or integer value, or a more complex expression (see section 2.5.2). In the example, we have a boolean 'or' expression that evaluates to the class of all things that are plants or that are parts of plant.

We introduce oil:Expression as a common placeholder, with oil:ConcreteTypeExpression and oil:ClassExpression as specialization placeholders. oil:BooleanExpression is introduced as a sibling of these two, since we want to be able to construct boolean expressions with either kind of expression. The specific boolean operators, 'and', 'or' and 'not, are introduced as subclasses. Also, notice that since a single class is essentially a simple kind of class expression, rdfs:Class itself should be a subclass of oil:ClassExpression (see figure 2.7).

The 'and', 'or' and 'not' operators are connected to operands using the oil:hasOperand property. This property again has no direct equivalent in OIL primitive terms, but is a helper to connect two class expressions, because in the RDF data model one can only relate two classes by means of a Property.

In our example, we need to serialize a boolean 'or'. The RDF Schema definition of the operator looks like this:

```
<rdfs:Class rdf:ID="Or">
  <rdfs:subClassOf rdf:resource="#BooleanExpression"/>
</rdfs:Class>
```

and the helper property is defined as follows:

```
<rdf:Property rdf:ID="hasOperand">
  <rdfs:domain rdf:resource="#BooleanExpression"/>
  <rdfs:range rdf:resource="#ClassExpression"/>
</rdf:Property>
```

The fact that hasOperand is only to be used on boolean class expressions is expressed using the rdfs:domain construction. This type of modeling stems directly from the RDF property-centric approach.

Now we apply what we defined above to the example:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass>
        <oil:Or>
          <oil:hasOperand rdf:resource="#plant"/>
          <oil:hasOperand>
            <HasValue>
              <oil:onProperty
                rdf:resource="#is-part-of"/>
              <oil:toClass rdf:resource="#plant"/>
            </HasValue>
          </oil:hasOperand>
        </oil:Or>
      </oil:toClass>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

Observe that the HasValue property restriction is not related to the class by a hasPropertyRestriction property, but by a hasOperand property. This stems from the fact that the property restriction plays the role of a boolean operand here.

Lists of statements

Now, we illustrate some more features by translating the second class definition, "elephant".

The first bit is trivial:

```
<rdfs:Class rdf:ID="elephant"> </rdfs:Class>
```

Next, we need to translate the OIL subsumption statement to RDFS. In this statement, a list of superclasses is given. In the RDFS syntax, we model these as seperate subClassOf statements:

```
<rdfs:Class rdf:ID="elephant">
  <rdfs:subClassOf rdf:resource="#mammal"/>
  <rdfs:subClassOf rdf:resource="#herbivore"/>
</rdfs:Class>
```

Next, we have two slot constraints. The first of these is a value-type restriction, and it is serialized in the same manner as we showed in the ”herbivore” example:

```
<rdfs:Class rdf:ID="elephant">
  <rdfs:subClassOf rdf:resource="#mammal"/>
  <rdfs:subClassOf rdf:resource="#herbivore"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass rdf:resource="#plant"/>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

Slot constraints to concrete types

The second slot constraint is a restriction to a particular concrete type. In OIL, a shortcut syntax for such restrictions has been introduced in the form of a ”has-filler” primitive. We serialize this like we do with the other slot constraints: we introduce a class oil:HasFiller and helper properties, oil:stringFiller and oil:integerFiller, to connect to the value:

```
<oil:HasFiller>
  <oil:onProperty rdf:resource="#colour"/>
  <oil:stringFiller>grey</oil:stringFiller>
</oil:HasFiller>
```

In RDF(S), there is unfortunately no direct way to constrain the value of a property to a particular datatype. Therefore, the range value of oil:stringFiller can not be constrained to contain only strings. Only for clarity we created two subclasses of rdfs:Literal, named oil:String and oil:Integer.

```
<rdfs:Class rdf:ID="String">
   <rdfs:comment>
   The subset of Literals that are strings.
   </rdfs:comment>
   <rdfs:subClassOf
      rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Class>
```

The range of the filler properties can now be set to the appropriate class, although it is still possible to use any type of Literal. The semantics of rdfs:Literal are only that anything of this type is atomic, i.e. it will not be processed further by an RDF processor. The fact that in this case it should be a string value can only be made an informal guideline.

```
<rdf:Property ID="stringFiller">
  <rdfs:domain rdf:resource="#HasFiller"/>
  <rdfs:range rdf:resource="#String"/>
</rdf:Property>
```

Using all this, we get the following complete translation of the class ”elephant”:

```
<rdfs:Class rdf:ID="elephant">
  <rdfs:subClassOf rdf:resource="#mammal"/>
  <rdfs:subClassOf rdf:resource="#herbivore"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass rdf:resource="#plant"/>
    </oil:ValueType>
    <oil:HasFiller>
      <oil:onProperty rdf:resource="#colour"/>
      <oil:stringFiller>grey</oil:stringFiller>
    </oil:HasFiller>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

Observe that it is allowed to have more than one property restriction within the hasPropertyRestriction element.

An additional observation that can be made is that the above OIL class can still be partially interpreted by an RDFS processor. It will recognize that ”elephant” is a class and that it is a subclass of both ”mammal” and ”herbivore”. This is a correct but incomplete interpretation of the meaning of the class: it will not recognize the meaning of the property restrictions.

Conclusion

The serialization we propose gives us enough expressiveness to translate any possible OIL class definition to an RDF syntax. Use of RDF(S) specific constructs is maximized without sacrificing clarity of the specification, to enable RDF agents that are not OIL-aware to understand as much of the specification as possible, while retaining the possibility to translate back to OIL unambiguously. This achieves our goal of maximising sharing of metadata, as stated in section 2.1.

In the next section, we will examine how to serialize global slot definitions.

2.5.3 Slot definitions

Both OIL and RDFS allow slots as first-class citizens of an ontology. Therefore, slot definitions in OIL map nicely onto property definitions in RDFS. Furthermore, the ”subslot-of”, ”domain”, and ”range” properties have also almost direct equivalents in RDFS. In table 2.2, an overview of the OIL constructs and the corresponding RDFS constructs is given.

There are a few subtle differences between domain and range restrictions in OIL and their equivalents in RDFS. In OIL, multiple domain and range restrictions on a single slot are allowed. The interpretation of such a set of restrictions is the *intersection* of the classes in the individual statements (conjunctive semantics). In RDFS, multiple domain statements are allowed, but their interpretation is the *union* of the classes in the statements (disjunctive semantics). This limits the reasoning capabilities of RDFS drastically[8].

[8]For example, it is never possible to derive class membership from a domain statement when union semantics are used.

Table 2.1: Class-definitions in OIL and the corresponding RDF(S) constructs

OIL primitive	RDFS syntax	type
class-def	`rdfs:Class`	class
subclass-of	`rdfs:subClassOf`	property
class-expression	*`oil:ClassExpression`* (placeholder only)	class
and	`oil:And` (subclass of BooleanExpression)	class
or	`oil:Or` (subclass of BooleanExpression)	class
not	`oil:Not` (subclass of BooleanExpression)	class
slot-constraint	*`oil:PropertyRestriction`* (placeholder only)	class
	`oil:hasPropertyRestriction` (rdf:type of rdfs:ConstraintProperty)	property
	`oil:CardinalityRestriction` (placeholder only) (subclass of oil:PropertyRestriction)	class
has-value	`oil:HasValue` (subclass of oil:PropertyRestriction)	class
has-filler	`oil:HasFiller` (subclass of oil:PropertyRestriction)	class
value-type	`oil:ValueType` (subclass of oil:PropertyRestriction)	class
max-cardinality	`oil:MaxCardinality` (subclass of oil:CardinalityRestriction)	class
min-cardinality	`oil:MinCardinality` (subclass of oil:CardinalityRestriction)	class
cardinality	`oil:Cardinality` (subclass of oil:CardinalityRestriction)	class

Despite these semantics for domain, a Property can have at most one range restriction in RDFS. However, the current consensus within the RDF community is that the semantics of domain and range should change in the next release of RDFS.[9] We anticipated on such a change, and interpret both multiple domain and multiple range restrictions with conjunctive semantics.

Another difference with RDFS is that OIL not only allows classes as range and domain of properties, but also class-*expressions*, and – for range – concrete-type expressions. It is not possible to reuse rdfs:range and rdfs:domain for these sophisticated expressions, because of the conjunctive semantics of multiple range statements: we cannot

[9] According to discussions on the `rdf-interest` and `rdf-logic` mailinglists.

extend the range of rdfs:range or rdfs:domain, we can only restrict it. In our RDFS serialization of OIL, we therefore introduce two new ConstraintProperties oil:domain and oil:range. They have the same domain as their RDFS equivalent (i.e., rdf:Property), but have a broader range. For domain, class expressions are valid fillers, for range both class expressions and concrete type expressions may be used:

```
<rdfs:ConstraintProperty rdf:ID="domain">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/
                        22-rdf-syntax-ns#Property"/>
  <rdfs:range rdf:resource="#ClassExpression"/>
</rdfs:ConstraintProperty>

<rdfs:ConstraintProperty rdf:ID="range">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/
                        22-rdf-syntax-ns#Property"/>
  <rdfs:range rdf:resource="#Expression"/>
</rdfs:ConstraintProperty>
```

When translating a slot definition, rdfs:domain and rdfs:range should be used for simple (one class) domain and range restrictions. For example:

slot-def *gnaws*
subslot-of *eats*
domain Rodent

will be translated into:

```
<rdf:Property rdf:ID="gnaws">
  <rdfs:subPropertyOf rdf:resource="#eats"/>
  <rdfs:domain rdf:resource="#Rodent"/>
</rdf:Property>
```

For more complicated statements the oil:range or oil:domain properties should be used:

slot-def *age*
domain (elephant **or** lion)
range (**range** 0 70)

is in the RDFS representation:

```
<rdf:Property rdf:ID="age">
  <oil:domain>
    <oil:Or>
      <oil:hasOperand rdf:resource="#elephant"/>
      <oil:hasOperand rdf:resource="#lion"/>
    </oil:Or>
  </oil:domain>
  <oil:range>
    <oil:Range>
      <oil:integerValue>0</oil:integerValue>
      <oil:integerValue>70</oil:integerValue>
    </oil:Range>
  </oil:range>
</rdf:Property>
```

To specify that the range of a property is string or integer, we use our definitions of oil:String and oil:Integer as subclasses of rdfs:Literal. For example, to state that the range of age is integer, one could say:

```
<rdf:Property ID="age">
  <rdfs:range rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#Integer">
</rdf:Property>
```

However, global slot-definitions in OIL allow specification of more aspects of a slot than property definitions in RDFS do. Besides the domain and range restrictions, OIL slots can also have an ”inverse” attribute and qualities like ”transitive” and ”symmetric”.

We therefore add a property ”inverseRelationOf” with ”rdf:Property” as domain and range. We also add the classes ”TransitiveProperty”, ”FunctionalProperty” and ”SymmetricProperty” to reflect the different qualities of a slot. In the RDFS serialization of OIL, the rdf:type property can be used to add a quality to a property. For example, the OIL definition of:

slot-def *has-part*
inverse *is-part-of*
properties transitive

is in RDFS:

```
<rdf:Property rdf:ID="has-part">
  <rdf:type rdf:resource=
    "http://www.ontoknowledge.org/oil/rdf-schema/
     2000/11/10-oil-standard#TransitiveProperty"/>
  <oil:inverseRelationOf rdf:resource="#is-part-of"/>
</rdf:Property>
```

or, in the abbreviated syntax:

```
<oil:TransitiveProperty rdf:ID="has-part">
  <oil:inverseRelationOf rdf:resource="#is-part-of"/>
</oil:TransitiveProperty>
```

This way of translating the qualitities of properties features the same nice object-meta distinction (between the OIL language on the one hand and the actual ontology on the other hand) as the translation of the ”type” of a class (see section 2.5.2). In an actual ontology, the property ”has-part” can be considered as an *instance* of a TransitiveProperty. It is allowed to make a property an instance of more than one class, and thus giving it multiple qualities. Note that this way of representing qualities of properties in RDFS follows the proposed general approach of modeling axioms in RDFS, presented in [Staab et al., 2000]. In this approach, the same distinction between language-level constructs and schema-level constructs is made.

One alternative way of serializing the attributes of properties would be to define the qualities ”transitive” and ”symmetric” as subproperties of rdf:Property. Properties in the actual ontology (e.g. ”has-part”) would in their turn be defined as subProperties of these qualities (e.g. transitiveProperty). However, this would mixup the use of properties at the OIL-specification level and at the actual ontology level.

A third way would be to model the qualities as subproperties of rdf:Property again, but to define properties in the actual ontology as instances (rdf:type) of such qualities. In this aproach, the object-meta level distinction is preserved. However, we dislike the use of rdfs:subPropertyOf at the meta-level, because then rdfs:subPropertyOf has two meanings, at the meta-level and at the object-level.

In our opinion, the first solution is preferable, because of the clean distinction it makes between the meta and object level.

Table 2.2: Slot-definitions in OIL and the corresponding RDF(S) constructs.

OIL primitive	RDFS syntax	type
slot-def	`rdf:Property`	class
subslot-of	`rdfs:subPropertyOf`	property
domain	`rdfs:domain`	property
	`oil:domain`	property
range	`rdfs:range`	property
	`oil:range`	property
inverse	`oil:inverseRelationOf`	property
transitive	`oil:TransitiveProperty`	class
functional	`oil:FunctionalProperty`	class
symmetric	`oil:SymmetricProperty`	class

2.5.4 Axioms

Axioms in OIL are factual statements about the classes in the ontology. They correspond to n-ary relations between class expressions, where n is 2 or greater.

RDF only knows binary relations (properties). Therefore, we cannot simply map OIL axioms to RDF properties. Instead, we chose to model axioms as classes, with helper properties connecting them to the class expressions involved in the relation. Since axioms can be considered objects, this is a very natural approach towards modeling them in RDF (see also [Staab and Mädche, 2000, Staab et al., 2000]). Observe also that binary relations (properties) are modeled as objects in RDFS as well (i.e., any property is an instance of the class rdf:Property). We simply introduce a new primitive *alongside* rdf:Property for relations with higher arity (see figure 2.7).

We introduce a placeholder class oil:Axiom, and model specific types of axioms as subclasses:

```
<rdfs:Class ID="Disjoint">
  <rdfs:subClassOf rdf:resource="#Axiom"/>
</rdfs:Class>
```

and likewise for Equivalent.

We also introduce a property to connect the axiom object with the class expressions it relates to each other: oil:hasObject is a property connecting an axiom with an object

class expression. For example, to serialize the axiom that herbivores, omnivores and carnivores are (pairwise) disjoint:

```
<oil:Disjoint>
  <oil:hasObject rdf:resource="#herbivore"/>
  <oil:hasObject rdf:resource="#carnivore"/>
  <oil:hasObject rdf:resource="#omnivore"/>
</oil:Disjoint>
```

Since in a disjointness axiom (or an equivalence axiom) the relation between the class expressions is bidirectional, we can connect all class expressions to the axiom object using the same type of property.

However, in a covering axiom (like cover or disjoint-cover), the relation between class expressions is not bidirectional: one class expression plays the role of covering, several other class expressions play the role of being part of that covering.

For modeling covering axioms, we introduce a seperate placeholder class, oil:Covering, which is a subclass of oil:Axiom. The specific types of coverings available are modeled as subclasses of oil:Covering again:

```
<rdfs:Class ID="Cover">
  <rdfs:subClassOf rdf:resource="#Covering"/>
</rdfs:Class>

<rdfs:Class ID="DisjointCover">
  <rdfs:subClassOf rdf:resource="#Covering"/>
</rdfs:Class>
```

Furthermore, two additional properties are introduced: oil:hasSubject, to connect a covering axiom with its subject, and oil:isCoveredBy, which is a subproperty of oil:hasObject, to connect a covering axiom with the classes that cover the subject.

For example, we serialize the axiom that the class animal is covered by carnivore, herbivore, omnivore, and mammal (i.e. every instance of animal is also an instance of at least one of the other classes).

```
<oil:Cover>
  <oil:hasSubject rdf:resource="#animal"/>
  <oil:isCoveredBy rdf:resource="#carnivore"/>
  <oil:isCoveredBy rdf:resource="#herbivore"/>
  <oil:isCoveredBy rdf:resource="#omnivore"/>
  <oil:isCoveredBy rdf:resource="#mammal"/>
</oil:Cover>
```

2.5.5 Restrictions to valid expressions

In the previous sections we have shown how the knowledge representation constructs in OIL can be defined as an extension to RDF Schema. With these constructs, every OIL ontology can be fully expressed in an RDF Schema representation. However, it was not possible to define the extension in such a way that all schemas that follow it are also valid OIL ontologies. In other words, there are some restrictions to valid ontologies that are not expressible in the RDF Schema extension.[10]

[10] With "valid" we mean: not allowed by the BNF grammer of OIL. From the logical point of view, there's nothing wrong with a statement like (dog **and** (**min** 0)), it just happens to be equivalent to the empty class.

First, there is a problem with datatypes. It cannot be enforced that instances of oil:String are really strings or that instances of oil:Integer are really integers. Consequently, it is syntactically possible to state:

```
<rdf:Property rdf:ID="weight">
  <rdf:range>
    <oil:Min>
      <oil:integerValue>nonsense</oil:integerValue>
    </oil:Min>
  </rdf:range>
</rdf:Property>
```

This is due to the fact that the RDF Schema specification has (intentionally) not specified any primitive datatypes. According to the specification, the work on data typing in XML itself should be the foundation for such a capability.

Second, the RDF Schema specification of OIL does not prevent the intertwining of boolean expressions of classes with boolean expressions of concrete data types. Although a statement like (dog **and** (**min** 0)) is not allowed in OIL, it is syntactically possible to state:

```
<oil:And>
  <oil:hasOperand rdf:resource="#Dog">
  <oil:hasOperand>
    <oil:Min>
      <oil:integerValue>0</oil:integerValue>
    </oil:Min>
  </oil:hasOperand>
</oil:And>
```

To prevent this kind of mixing, we could have introduced separate boolean operators for class expressions and concrete type expressions, but in our opinion, this would have made the schema too convoluted.

Finally, another kind of problem is that the schema cannot prevent the unnecessary use of the OIL variants of standard RDF Schema constructs, like oil:subClassOf, oil:range and oil:domain. Although this unnecessary use does not affect the semantics of the ontology, it limits the compatibility of ontologies with plain RDF Schema.

2.6 Compatibility with RDF Schema

In this section we will discuss the extent of the compatibility that we have achieved between the semantic extension (OIL), and the underlying language (RDF Schema).

As for any ontology language, we can distinguish three levels: First, the ontology language, the language in which to state class-definitions, subclass-relations, attribute-definitions etc., for example OIL. Second, the ontological classes, for example the classes ”giraffe” or ”herbivore”, their subclass relationships, and their properties (such as eats). These are of course expressed in the language of the first level. Third, the instances of the ontology, such as individual giraffes or lions that belong to classes defined at the second level.

If we look at the existing W3C RDF/RDF Schema recommendation, these levels have the following form:

1. the ontology language is of course RDF Schema;
2. specific classes, their properties and relations are therefore written in RDF Schema, eg:

```
<rdfs:Class rdf:ID="herbivore">
  <rdfs:subClassOf rdf:resource="#animal">
</rdfs:Class>
<rdf:Property rdf:ID="eats"/>
```

3. instances are written in RDF (note: *not* RDF Schema), eg:

```
<rdf:Description rdf:about="http://www.cs.vu.nl/~frankh">
  <rdf:type rdf:resource="#herbivore"/>
</rdf:Description>
```

If we consider a semantic extension of RDF Schema such as OIL, the situation is as follows:

1. The ontology language is OIL, but it is important to realise that OIL includes RDF Schema as a sublanguage
2. As a result, class expressions written in OIL are actually also legal RDF Schema. For example, besides being a meaningful OIL definition, the class definition of "herbivore" in item 2 above is also a legal example of an RDF Schema definition. Of course, since OIL is an *extension* of RDF Schema, not all parts of an OIL definition are *meaningful* RDF Schema. For example, in

```
<rdfs:Class rdf:ID="herbivore">
  <rdfs:subClassOf rdf:resource="#animal"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass>
        <oil:Or>
          <oil:hasOperand rdf:resource="#plant"/>
          <oil:hasOperand>
            <oil:HasValue>
              <oil:onProperty
                rdf:resource="#is-part-of"/>
              <oil:toClass rdf:resource="#plant"/>
            </oil:HasValue>
          </oil:hasOperand>
        </oil:Or>
      </oil:toClass>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

   the semantics of the hasPropertyRestriction statement will not be interpretable by an RDF Schema processor. The entire state is legal RDF syntax, so it can be parsed, but the intended semantics of the property restriction itself can only be understood by an OIL-aware application. Notice that the first subClassOf statement is still fully interpretable even by an OIL-unaware RDF Schema processor.

3. OIL instances are written as RDF! This is an important consequence of the fact that the second level is organised as an extension of RDF Schema.

The above shows that we have now achieved two important compatibility results: first of all, OIL is *backward compatible* with RDF Schema, i.e. every RDF Schema specification is also a valid OIL ontology declaration. Second, we have achieved *partial forward compatibility*, i.e. even if an ontology is written in the richer modelling language (OIL), a processor for the simpler ontology language (RDF Schema) can still:

a) fully interpret all the instance information of the ontology, and

b) partially interpret the class-structure of the ontology. This can be achieved by simply ignoring any statement not from the rdf or rdfs namespaces (in our example those from the oil namespace). For example, in the above definition of ”herbivore”, an RDF Schema processor will interpret this statement simply as stating that herbivores are a subclass of animals, and that they some other property that it cannot interpret. This is a correct albeit partial interpretation of the definition.

Such partial interpretability of semantically rich meta-data by semantically poor processing agents is a crucial step towards the sharing of meta-data on the Semantic Web. We cannot realistically hope that all of the Semantic Web will be built on a single standard for semantically rich meta-data. The above shows that multiple semantic modelling languages do not have to lead to meta-data that are totally uninterpretable by others. Instead, simpler processors can still pick up as much of the meta-data from rich processors as they can ”understand”, and safely ignore the rest in the knowledge that their partial interpretation is still correct with respect to the original intention of the meta-data.

2.7 Related work

Work on ontology representation languages dates back to the work on frame-languages in the early days of AI. The origins of RDF, OWL and description logics in general can be traced back to early work on semantic nets and more in particular on the KL-One Knowledge Representation System [Brachman and Schmolze, 1985] and its successor CLASSIC [Borgida et al., 1989], and efforts such as Telos [Mylopoulos et al., 1990].

However, efforts of designing ontology-representation languages that are Web-enabled only date from recent years. The most prominent efforts in this area have been SHOE [Luke et al., 1996, Heflin and Hendler, 2000], Ontobroker [Fensel et al., 1998, Decker et al., 1999], OIL and DAML-ONT[11], and more recently, as a replacement for DAML-ONT, DAML+OIL[12] and its successor, the Web Ontology Language OWL [Dean and Schreijber, 2004].

DAML-ONT shares with our own proposal the principle that an ontology language should maintain maximum backwards compatibility with existing web standard languages, and in particular RDF Schema. The difference between OIL and DAML-ONT lies in the degree to which the languages succeed in maximising the ontological content that can be understood by an “RDF Schema agent” (ie. an application that understands

[11] DAML-ONT Initial Release, `http://www.daml.org/2000/10/daml-ont.html`

[12] DAML+OIL `http://www.daml.org/2000/12/daml+oil-index.html`

RDF Schema but does not recognise the language specific extensions, OIL or DAML-ONT). Unlike OIL, DAML-ONT is built on top of RDFS in a way that allows little if any ontological content to be understood by an RDFS agent. In OIL, for example, stating simple subclass relationships between classes is done using the RDFS subClassOf property:

```
<rdfs:Class ID="Male">
  <rdfs:subClassOf rdf:resource="#Animal"/>
</rdfs:Class>
```

This part of OIL ontologies is therefore accessible to any RDFS agent. In contrast, DAML-ONT uses its own locally defined "subClassOf" property, for example:

```
<daml:Class ID="Male">
  <daml:subClassOf resource="#Animal"/>
</daml:Class>
```

The DAML-ONT subClassOf property is then defined to be "equivalent to" `rdfs:subClassOf`, but the definition of `daml:equivalentTo` itself relies cyclicly on the definition of `daml:sub-PropertyOf`. Therefore even simple subclass relationships in a DAML ontology are inaccessible to an RDFS agent. The situation is even worse when it comes to more complex class definitions. For example, the definition of the class `TallMan` as the intersection of the classes `Man` and `TallThing` is expressed in DAML-ONT as follows:

```
<daml:Class rdf:ID="TallMan">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#TallThing"/>
    <daml:Class rdf:about="#Man"/>
  </daml:intersectionOf>
</daml:Class>
```

This is completely opaque to an RDFS agent as it will not understand the semantics of `daml:intersectionOf` In OIL, the definition of `TallMan` would rely on the fact that intersection is implicit in the semantics of `rdfs:subClassOf`:

```
<rdfs:Class ID="TallMan">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
    rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#TallThing"/>
  <rdfs:subClassOf rdf:resource="#Man"/>
</rdfs:Class>
```

making the sub-class relationships accessible to any RDF Schema agent. In conclusion, we argue that:

- OIL and DAML-ONT are currently the only two web-based ontology languages that are built on top of RDFS;
- of these, OIL achieves a much larger degree of "backward compatability" with RDF.

DAML+OIL is a proposal for an ontology language developed by a joint European-US team of researchers. The proposal merges the ideas incorporated in DAML-ONT with those in OIL. Specifically, some of the ideas, presented in this article, on how to represent a KR language in RDFS have been adopted by DAML+OIL. In effect, DAML+OIL is "backward compatible" with RDF to a much larger degree than the initial DAML-ONT language was, although still not quite as much as the original OIL specification.

Finally, OWL is the official successor of DAML+OIL. The World Wide Web Consortium has made minimal changes to the original DAML+OIL proposal and turned this into an official Recommendation for ontology description. It is gratifying to see that many of the lessons learned with the design of OIL have found their way into this proposal, although again, the compatibility between RDFS and the KR formalism modeled on top is less than was intended in the original OIL spec. Illustrative to this point is the fact that OWL still has separate notions of `owl:Class` and `owl:intersectionOf`, instead of reusing RDF Schema primitives as shown in the example above.

2.8 Conclusion

In this chapter, we have shown why a machine-accessible representation of the information available on the Web is both useful and necessary. We have introduced the Resource Description Framework and RDF Schema, and we have also shown that for many purposes RDF Schema is only a small step towards the required expressiveness. Finally, we have illustrated how, by extending RDF Schema with additional modeling primitives as defined by a more formal knowledge representation scheme, such as OIL, RDF Schema can still play an important role as a carrier language.

An important advantage of our approach is the maximization of the compatibility with RDFS: not only is every RDF Schema document a valid OIL ontology declaration, but every OIL ontology can be partially interpreted by a semantically poorer processing agent. This partial interpretation will of course be incomplete, but correct under the intented semantics of the ontology. We firmly believe that our way of extending is generally applicable across knowledge representation formalisms.

Finally, we have given a brief overview of the history of the different web ontology languages, to illustrate how the ideas illustrated in this chapter have found their way into the latest official standard for ontology representation, OWL.

Chapter 3

Query Languages for the Semantic Web

In the previous chapter, we have seen the introduction of several formalisms for knowledge representation on the Semantic Web. To access this knowledge, however, a *query language* is an essential tool. In this chapter, we will define several requirements for such a query language, and we will see if and how several existing XML and RDF query languages deal with these requirements.

The work presented here was originally published in [Broekstra et al., 2000] and [Haase et al., 2004].

3.1 Introduction

Although a representation formalism is an essential building block for a 'Knowledge Web', representing information in a machine-accessible way alone is not enough. Enabling querying is of course just as important. This is where query languages come into focus. Many query languages already exist. The most obvious example is SQL, the standard query language for relational databases.

We believe that defining a suitable RDF query language should be a top priority in terms of standardization. Querying is a fundamental functionality required in almost any Semantic Web application. Judging from the impact of SQL to the database community, standardization will definitely help the adoption of RDF query engines, make the development of applications a lot easier, and will thus help the Semantic Web in general.

In this chapter, we will explore what properties a query language for semistructured data should have, and what the difference is with existing approaches such as SQL.We will then discuss several proposals for query languages.

The chapter is organized as follows. Section 3.2 identifies several general requirements for query languages that deal with semi-structured data. In section 3.3 we briefly look at several proposals for XML query languages. Section 3.5 then goes further into

query language requirements and properties, explicitly for RDF query languages. Section 3.5.3 introduces the six RDF query languages that are compared along with the implementations that we have used for testing. Section 3.6 demonstrates RDF query use cases, which are grouped into several categories. These use cases are used to compare the individual languages and expose the set of features supported by each language. Section 3.7 presents a wish list for further important but yet unsupported query language features. We conclude in Section 3.8 with a summary of our results.

3.2 General properties of Query Languages

We can identify several general properties with which one can characterize query languages. Here, we name six such properties, which we will use throughout the rest of this chapter to try and characterize strong (and weak) points of the languages that we discuss.

- *Expressiveness* Expressiveness indicates how powerful queries can be formulated in a given language. Ideally, a query languae should be expressive enough to allow the retrieval of any arbitrary combination of values from the queried model, that is, be *complete* with respect to its datamodel. Usually, expressiveness is restricted to maintain other properties such as safety and to allow an efficient (and optimizable) execution of queries.

- *Closure* The closure property requires that the results of an operation are again elements of the data model. This means that if a query language operates on a graph data model, the query results would again have to be graphs.

- *Adequacy* A query language is called adequate if it allows access to all concepts of the underlying data model. This property therefore complements the closure property: For the closure, a query result must not be outside the data model, for adequacy the entire data model needs to be exploited.

- *Orthogonality* The orthogonality of a query language requires that all operations may be used independently of the usage context.

- *Safety* A query language is considered safe, if every query that is syntactically correct returns a finite set of results (on a finite data set). Typical concepts that cause query languages to be unsafe are recursion and negation.

3.2.1 Path expressions

One of the main distinguishing features of query languages for semistructured data is their ability to reach to arbitrary depths in the data graph. To do this, these languages all use the notion of path expressions. A path expression is a simple query, the result of which, for a given data graph, is a set of labels of nodes and/or edges of the graph. For example, consider the following bit of XML:

```
<?xml version="1.0"?>
<body>
  This page is written by
  <author>Frank van Harmelen</author>.
  <location>
    His tel.nr. at work is <tel>47731</tel>,
    his number at home is <tel>555722</tel>, and his
    room number is <room>T3.57</room>.
  </location>
</body>
```

The result of the path expression `body.location.tel` would be the set of nodes with the associated values '47731', '555722'.

Many useful regular expressions can be used in path expressions to facilitate more complex expressions than just specification of the complete path. For example, a regular expression `location|name` specifies either a location node or a name node. Another useful pattern is the wildcard, which matches any node label. Using the symbol to express this (cf. [Abiteboul et al., 1999]), `body.tel` matches any path consisting of a body node followed by any node, followed by a tel node. Also, closure operations, like arbitrary repeats of a regular expression can be used. For example, `body*.tel` specifies the set of tel nodes that occur at arbitrary depth within the body node. At another level of abstraction, regular expressions can also be used to express matches on the actual string format of labels. For example the regular expression `body."[aA]uthor"` matches any author node within the body, possibly with the first letter capitalized.

Path expressions, although they are an essential feature of query languages for semistructured data, can only return a subset of nodes in the database. They can not construct new nodes, perform joins, or test values stored in the database. In other words: path expressions are necessary but not sufficient for a good query language on semistructured data. A query language that lacks path expressions can not be considered adequate, nor sufficiently expressive for querying semistructured data.

3.2.2 Why not just SQL?

For strictly relational data (as opposed to semistructured data), SQL is by far the most widely supported query language, including support for large data-storage, efficient indexing schemes, query-optimisers, etc. It would therefore be attractive if we could use this robust and widely available technology for our purposes of querying semistructured data . Unfortunately, this can only be done at the cost of a large gap between the data-model in the repository (e.g. RDF) and the data-model on which the query-language is based (the relational model).

To exemplify this, let us look at how the scenario would look for an XML implemetation in a relational-database: as a first step, we would have to encode the XML data-model in the relational model. This would be possible by assigning each node in an XML-tree a unique identifier, with each entry in the relational database linking such a node with all its descendants and attributes. The problems start when we want to use this as the basis for querying the XML-structure: each XML-query should be compiled into an SQL-query on the underlying relational tables. Typically, a single XML-query (such as: 'return all descendants of a given node') must be compiled into a complicated set of

SQL queries. It is not even clear whether a finite set of SQL-queries could be generated for every reasonable XML query.

Although perhaps attractive as a short term solution, we feel that in the long run this is not an appropriate solution. Rather, techniques for large data-storage, indexing schemes, queryoptimers, etc. should be provided for the native data-model (be it XML or RDF), instead of relying on these techniques for a completely different data model.

3.3 Querying XML

In this section, we discuss several approaches for a query language for XML, and we will compare them with the general requirements for such a query language as presented in the previous section.

3.3.1 XSL

The Extensible Stylesheet Language (XSL) [Adler et al., 2000], is a proposal for a language to express stylesheets for XML documents. It is currently under development at the W3C. XSL is divided in two parts:

- XSLT, a transformation language
- XSL-FO, a set of Formatting Objects

We will focus on the first of these here.

XSLT maps an input data tree to an output data structure. Although its primary role is to allow users to write transformations from XML to HTML describing the presentation of the XML document (i.e. it serves as a stylesheet mechanism), it can also serve in the role of query language. An XSL program is a set of template rules that are executed in a best-match manner by recursively traversing over the nodes in the input data tree. An example program, which retrieves the names of all animals in the zoo (see figure 3.1):

```
<xsl:template>
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="/zoo/*/animal/name">
  <result>
    <xsl:value-of/>
  </result>
</xsl:template>
```

Each of the two `xsl:template` constructs specifies a template rule. If we look at the second template rule, we see a `match` attribute, which specifies the pattern, and the body of the rule is the template. In the first rule, there is no match attribute, which means that the rule is matched by any node in the input data tree.

If we apply this program to the example XML database in figure 3.1, the first template rule is matched by the top level node `zoo`. The `xsl:apply-templates` directive states that all templates are to be executed on the contents of the zoo node. In this manner, the data tree is traversed recursively, until the search hits the name node that matches the

```
<zoo>
  <habitat name="Savannah">
    <attendant idref="att1" />
    <animal type="elephant">
      <name> Henri </>
      <gender> male </>
      <favorite_food> peanuts </>
    </animal>
    <animal type="penguin">
      <name> Tux </>
      <gender> male </>
      <favorite_food> fish </>
    </animal>
    <animal type="giraffe">
      <name> Andy </>
      <gender> male </>
      <favorite_food> leafs </>
    </animal>
  </habitat>
  <habitat name="Polar World">
    <animal type="penguin">
      <name> Frisky </>
      <gender> female </>
      <favorite_food> fish </>
    </animal>
  </habitat>
  <employee id="att1">
    <name> Frank </name>
  </employee>
</zoo>
```

Figure 3.1: An example XML database

more specific second template rule. This rule specifies that when matched, the output should be the value of the current node, enclosed in `result` tags.

Applied to the database in figure 3.1, we get the following result:

```
<result> Henri </result>
<result> Tux </result>
<result> Andy </result>
<result> Frisky </result>
```

As we can see, XSL has path expressions to allow matching, and uses a recursive descent mechanism to traverse the data model, fitting closely to the XML tree model. However, the things that XSL can not do include binding values to variables, and expressing joins, nor can it express any other boolean operations on sets. A further disadvantage is that the context of an XSL program (i.e., the XML document(s) which it should take as its input) is not specified in the program itself, which makes the input mechanism unclear.

Concluding, one could say that the stylesheet/transformation background of XSL, while offering much of the same functionality as a query language, makes it less useful for the purpose of querying. It lacks the required expressiveness, specifically. It is worthwile to note that XSL by the very nature of its recursive template matching approach is an unsafe language.

3.3.2 XQL

The XML Query Language (XQL) [Robie et al., 1998] is a notation for addressing and filtering the elements and text of XML documents. XQL is a natural extension to the XSL pattern syntax. According to [Robie et al., 1998], it enhances XSL with a.o. boolean operators, filters and indexing capabilities.

XQL extracts data by means of patterns and path expressions, just like we saw with XSL. However, XQL provides far easier syntax and more powerful selection mechanisms.

An example XQL query, in the context of our zoo:

```
/zoo/habitat
```

This returns all `habitat` elements in our zoo. To find all `habitat` elements anywhere in the zoo (i.e. at arbitrary depth in the tree):

```
/zoo//habitat
```

To find all penguins:

```
//animal[@type = penguin]
```

In this last example, we see how XQL differentiates attributes from elements (the prefix) and how one can filter using boolean expressions. Just like in XSL, in XQL variable binding is still not possible, and hence, no joins can be expressed. The authors claim that nevertheless XQL can be used to query over multiple documents, but this seems to assume that these documents are all available in a single XML repository. However, a mechanism for this is not prescribed by the XQL proposal. Also, XQL does not prescribe an output format. A direct result of this is that XQL can not guarantee compositionality of queries, in other words, the languages is not *closed*. It also means that XQL does not offer the ability to construct alternative views on data sources. Its usefulness seems to be strictly limited to data retrieval.

3.3.3 XML-QL

XML-QL [Deutsch et al., 1998, Abiteboul et al., 1999] combines XML syntax with query language techniques. It uses path expressions and patterns to extract data from the input XML data, has variables to which this data can be bound and has templates which show how the output XML data is to be reconstructed.

An example of an XML-QL query is:

```
where <habitat type=$T>
        <animal type="penguin">
          <name> $N </name>
          <gender> male </gender>
          <favorite_food> $F </favorite_food>
        </animal>
      </habitat> in "www.a.b.c/zoo.xml"
construct <result>
            <name> $N </name>
            <lives_in> $T </lives_in>
            <food> $F </food>
          </result>
```

As we can see, XML-QL is based on a where/construct syntax, instead of the familiar select/ from/where of SQL. The construct clause corresponds to select, and the where combines the from and where parts of the query, that is, the ranging of variables and some filtering. In the above query, $T and $N are variables and the XML structure in the where-clause is a pattern. The pattern is matched in all possible ways to the data and the variables are bound to the corresponding values in the matching cases. In the construct part, we see how XML-QL can be used to construct new XML data. The mechanism is simple: we just add a template to the construct clause and use bound variables from the where clause to fill the template. So, when we apply this query to the example database of figure 3.1, the result is:

```
<result>
  <name> Tux </name>
  <lives_in> Savannah </lives_in>
  <food> fish </food>
</result>
```

XML-QL can use path expressions, query across multiple data sources, and express joins:

```
where <*.animal> <name> $N1 </name> <*./animal>
in "zoo.xml",
<*.animal> <name> $N2 </name> <*./animal>
in "anotherzoo.xml",
$N1 = $N2
construct <result> $N1 </result>
```

This query yields all names that animals in different zoos share. To summarize: XML-QL combines relatively easy syntax with powerful query language notions. In fact, XML-QL is relationally complete, that is, its expressiveness is on par with SQL when applied to relational data. One particular feature that neither XSL nor XQL share is that it can express joins and thus combine information from multiple data sources.

3.3.4 XQuery

XQuery [Boag et al., 2005] is a W3C Working Draft for a standardized query language for XML. It is heavily inspired by both XML-QL and XQL and is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents. Since its main features are similar to those of XQL and XML-QL, we do not discuss the language here in detail, but instead refer the reader to [Boag et al., 2005].

3.4 The need for an RDFS Query Language

RDF documents and RDF schemata can be considered at three different levels of abstraction:

1. at the *syntactic level* they are XML documents.[1]

[1] Actually, this is not necessarily true; non-XML syntaxes for RDF exist, but XML is the most widely used syntax for RDF.

2. at the *structure level* they consist of a set of triples.

3. at the *semantic level* they constitute one or more graphs with partially predefined semantics.

We can query these documents at each of these three levels. We will briefly consider the pros and cons of doing so for each level in the next sections. This will lead us to conclude that RDF(S) documents should really be queried at the semantic level. We will also briefly discuss RQL, a language for querying RDF(S) documents at the semantic level, which has been implemented in the Sesame architecture.

3.4.1 Querying at the syntactic level

As we have seen in chapter 2, any RDF model (and therefore any RDF schema) can be written down in XML notation. It would therefore seem reasonable to assume that we can query RDF using an XML query language (for example, XQuery [Boag et al., 2005]).

However, this approach disregards the fact that RDF is not just an XML notation, but has its own data model that is different from the XML tree structure. Relationships in the RDF data model that are not apparent from the XML tree structure become hard to query.

As an example, consider the RDF model in figure 3.2.

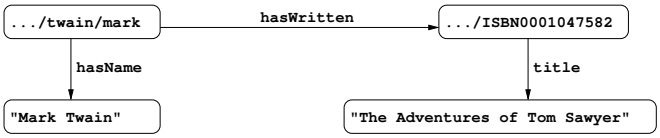


Figure 3.2: An example RDF graph

One possible serialization of the relations in XML looks like this:

```
<rdf:Description rdf:about="http://www.famouswriters.org/twain/mark">
    <s:hasName>Mark Twain</s:hasName>
    <s:hasWritten rdf:resource="http://www.books.org/ISBN0001047582"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.books.org/ISBN0001047582">
    <s:title>The Adventures of Tom Sawyer</s:title>
    <rdf:type rdf:resource="http://www.description.org/schema#Book"/>
</rdf:Description>
```

In an XML query language such as XQuery, expressions to traverse the data structure are tailored towards traversing a node-labeled tree. However, the RDF data model is a graph, not a tree, and moreover, both its edges (properties) and its nodes (subjects/objects) are labeled. In querying at the syntax level, this is literally left as an excercise for the query builder: one cannot query the relation between the resource signifying 'Mark Twain' and the resource signifying 'The Adventures of Tom Sawyer' without knowledge of the syntax that was used to encode the RDF data in XML.

Ideally, we would want to formulate a query like "Give me all the relationships that exist between Mark Twain and The Adventures of Tom Sawyer". However, using only the XML syntax, we are stuck with formulating an awkward query like "Give me all the elements nested in a Description element with an about attribute with value 'http://www.famouswriters.org/twain/mark', of which the value of its resource attribute occurs elsewhere as the about attribute value of a Description element that has a nested element title with the value 'The Adventures of Tom Sawyer'."

Not only is this approach inconvenient, it also disregards the fact that the XML syntax for RDF is not unique: the same RDF graph can be serialized in XML in a variety of ways. This means that one query will never be guaranteed to retrieve all the answers from an RDF model.

3.4.2 Querying at the structure level

When we abstract from the syntax, any RDF document represents a set of triples, each triple representing a statement of the form Subject-Predicate-Object. A number of query languages have been proposed and implemented that regard RDF documents as such a set of triples, and that allow to query such a triple set in various ways.

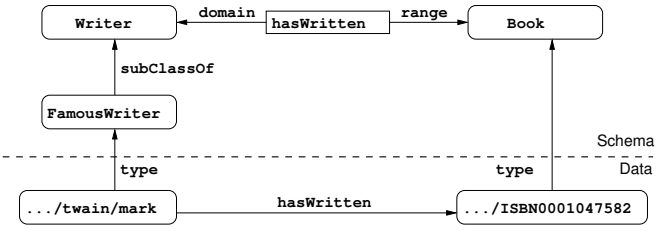


Figure 3.3: An example RDF Schema graph

Consider the example RDF Schema in figure 3.3. An RDF query language such as, for example, Squish [Miller, 2001] would allow us to query which resources are known to be of type FamousWriter:

```
SELECT ?x
FROM   somesource
WHERE  (rdf::type ?x FamousWriter)
```

The clear advantage of such a query is that it directly addresses the RDF data model, and that it is therefore independent of the specific syntax that has been chosen to represent the data.

However, a disadvantage of any query language at this level is that it interprets *any* RDF model only as a set of triples, including those elements which have been given a special semantics in RDFS. For example, since .../twain/mark is of type FamousWriter, and since FamousWriter is a subclass of Writer, .../twain/mark is also of type Writer, by virtue of the intended RDFS semantics of type and subClassOf. However, there is no triple that explicitly asserts this fact. As a result, the query

```
SELECT ?x
FROM   somesource
WHERE  (rdf::type ?x Writer)
```

will fail because the query only looks for explicit triples in the store, whereas the triple `(/twain/mark type Writer)` is not explicitly present in the store, but is implied by the semantics of RDFS.

3.4.3 Querying at the semantic level

What is clearly required is the means to query at the *semantic level*, that is, querying the full knowledge that a RDFS description entails and not just the explicitly asserted statements.

There are at least two options to achieve this goal:

1. Compute and store the closure of the given graph as a basis for querying.
2. Let a query processor infer new statements as needed per query.

While the choice of an RDF query language is, in principle, independent of the choice made in this respect, the fact remains that most RDF query languages have been designed to query a simple triple base, and have no specific functionality or semantics to discriminate between schema and data information.

3.5 Querying RDF

In the previous sections we have outlined some general requirement on query languages for semistructured data and have briefly discussed a few XML query languages. In this and following sections, we will focus on querying for RDF.

The Resource Description Framework (RDF) is considered to be the most relevant standard for data representation and exchange on the Semantic Web. The recent recommendation of RDF has just completed a major clean up of the initial proposal [Lassila and Swick, 1999] in terms of syntax [Beckett, 2004], along with a clarification of the underlying data model [Klyne and Carroll, 2004], and its intended interpretation [Hayes, 2004]. Several languages for querying RDF documents and have been proposed, some in the tradition of database query languages (i.e. SQL, OQL), others more closely inspired by rule languages. No standard for RDF query language has yet emerged, but the discussion is ongoing within both academic institutions, Semantic Web enthusiasts and the World Wide Web Consortium (W3C). The W3C recently chartered a working group[2] with a focus on accessing and querying RDF data.

In the rest of this chapter, we present a comparison of six representative query languages for RDF, highlighting their common features and differences along general dimensions for query languages and particular requirements for RDF. Our comparison does not claim to be complete with respect to the coverage of all existing RDF query languages. However, we try to maintain an up-to-date version of our report with an extended

[2] `http://www.w3.org/2001/sw/DataAccess/`

set of languages on our website [Haase et al., 2004]. This extended set of languages also includes RxPath[3] and RDFQL[4].

3.5.1 Related Work

Two previous tool surveys [Maganaraki et al., 2002, Fensel and Perez, 2002] and diverse web sites[5] have collected and compared RDF query languages and their associated prototype implementations. The web sites are usually focused on collecting syntactic example queries along several use cases. We follow this approach of illustrating a language but instantiate general categories of orthogonal language features to avoid the repetitiveness of use cases and capture a more extensive range of language features. Two tool surveys [Maganaraki et al., 2002],[Fensel and Perez, 2002] were published in 2002 and focused mainly only the individual prototype implementations comparing criteria like quality of documentation, robustness of implementation and to a minor extent the query language features[6] which changed tremendously in the past two years. We detail the feature set and illustrate supported features through example queries. [Gutierrez et al., 2004] analyzes the foundational aspects of RDF data and query languages, including computational aspects of testing entailment and redundancy.

It should be stressed that our comparison does not involve performance figures, as the focus is on the RDF query languages, not the tools supporting these languages.

3.5.2 Support for the RDF data model

The underlying data model directly influences the set of operations that should be provided by a query language. We therefore recapitulate the basic concepts of RDF and make note of their implications for the requirements on an RDF query language.

RDF abstract data model

The underlying structure of any RDF document is a collection of triples. This collection of triples is usually called the RDF graph. Each triple states a relationship (aka. edge, property) between two nodes (aka. resource) in the graph. This abstract data model is independent of a concrete serialization syntax. Therefore query languages usually do not provide features to query serialization-specific features, such as order of serialization.

Formal semantics and Inference

RDF has a formal semantics which provides a dependable basis for reasoning about the meaning of an RDF graph. This reasoning is usually called entailment. Entailment rules state which implicit information can be inferred from explicit information. Hence, RDF query languages can consider such entailment and can convey means to distinguish implicit from explicit data.

[3] `http://rx4rdf.liminalzone.org/RxPath`
[4] `http://www.intellidimension.com/`
[5] `http://www.w3.org/2001/11/13-RDF-Query-Rules/\#implementations`
[6] cf. [Maganaraki et al., 2002] for the most extensive summary

Support for XML schema data types

XML data types can be used to represent data values in RDF. XML Schema also provides an extensibility framework suitable for defining new datatypes for use in RDF. Data types should therefore be supported in an RDF query language.

Free support for making statements about resources

In general, it is not assumed that complete information about any resource is available in the RDF query. A query language should be aware of this and should tolerate incomplete or contradicting information.

3.5.3 RDF Query Languages

This section briefly introduces the query languages and actual systems that were used in our comparison.

RQL

RQL [Karvounarakis et al., 2002] is a typed language following a functional approach, which supports generalized path expressions featuring variables on both nodes and edges of the RDF graph. RQL relies on a formal graph model that captures the RDF modeling primitives and permits the interpretation of superimposed resource descriptions by means of one or more schemas. The novelty of RQL lies in its ability to smoothly combine schema and data querying while exploiting the taxonomies of labels and multiple classification of resources. RQL follows an OQL-like syntax: `select Pub from {Pub} ns3:year {y} where y = "2004" using namespace ns3 = ...`.

RQL is orthogonal, but not closed, as queries return variable bindings instead of graphs. However, RQL's semantics is not completely compatible with the RDF Semantics: a number of additional restrictions are placed on RDF models to allow querying with RQL[7].

RQL is implemented in ICS-FORTH's RDF Suite[8], and an implementation of a subset of it is available in the Sesame system[9]. For our evaluation we used Sesame version 1.0, which was released on March 25, 2004.

SeRQL

SeRQL [Broekstra and Kampman, 2003, Broekstra and Kampman, 2004] stands for Sesame RDF Query Language and is a querying and tranformation language loosely based on several existing languages, most notably RQL, RDQL and N3. Its primary design goals are unification of best practices from query language and delivering a lightweight yet expressive query language for RDF that addresses practical concerns.

[7] An example of such a restriction is that every property must have *exactly* one domain and range specified.
[8] `http://139.91.183.30:9090/RDF/`
[9] `http://www.openrdf.org/`

SeRQL syntax is similar to that of RQL though modifications have been made to make the language easier to parse. Like RQL, SeRQL is based on a formal interpretation of the RDF graph, but SeRQL's formal interpretation is based directly on the RDF Model Theory.

SeRQL supports generalized path expressions, boolean constraints and optional matching, as well two basic filters: select-from-where and construct-from-where. The first returns the familiar variable-binding/table result, the second returns a matching (optionally transformed) subgraph. As such, SeRQL construct-from-where-queries fulfill the closure and orthogonality property and thus allow composition of queries. SeRQL is not safe as it provides various recursive built-in functions.

SeRQL is implemented and available in the Sesame system, which we have used for our comparison in the version 1.0. A number of querying features are still missing from the current implementation. Most notable of these are functions for aggregation (minimum, maximum, average, count) and query nesting. In chapter 4, we will describe the SeRQL language in more detail.

TRIPLE

The term Triple denotes both a query and rules language as well as the actual runtime system [Sintek and Decker, 2001]. The language is derived from F-Logic [Kifer et al., 1995]. RDF triples `(S,P,O)` are represented as F-Logic expressions `S[P->O]`, which can be nested. For example, the expression `S[P1->O1, P2->O2[P3->O3]]` corresponds to three RDF triples `(S,P1,O1)`, `(S,P2,O2)`, and `(O2,P3,O3)`.

Triple does not distinguish between rules and queries, which are simply headless rules, where the results are bindings of free variables in the query. For example, `FORALL X <- ( X[rdfs:label->"foo"] )@default:ln.` returns all resources which have a label "foo". Since the output is a table of variables and possible bindings, Triple does not fulfill the closure property. Triple is not safe in the sense that it allows unsafe rules such as `FORALL X ( X[rdfs:label->"foo"] <- ( a[rdfs:label->"foo"] )@default:ln.`. While Triple is adequate and closed for its own data model, the mapping from RDF to Triple is not lossless. For example, anonymous RDF nodes are made explicit. Triple is able to deal with several RDF models simultaneously, which are identified via a suffix @model.

Triple does not encode a fixed RDF semantics. The desired semantics have to be specified as a set of rules along with the query. Datatypes are not supported by Triple. For the comparison, we used Triple in the latest version from March 14th, 2002 along with XSB 2.5 for Windows.

RDQL

RDQL currently has the status of a W3C submission [Seaborne, 2004].

The syntax of RDQL follows a SQL-like select pattern, where a from clause is omitted. For example, `select ?p where (?p, <rdfs:label>, "foo" )` collects all resources with label "foo" in the free variable p. The select clause at the beginning of the query allows projecting the variables. Namespace abbreviations can be defined in a

query via a separate "using" clause. RDF Schema information is not interpreted. Since the output is a table of variables and possible bindings, RDQL does not fulfill the closure and orthogonality property. RDQL is safe and offers preliminary support for datatypes.

For the comparison, we worked with Jena 2.0 of August 2003.

N3

Notation3 (N3) provides a text-based syntax for RDF. Therefore the data model of N3 conforms to the RDF data model. Additionally, N3 allows to define rules, which are denoted using a special syntax, for example: `?y rdfs:label "foo" => ?y a :QueryResult` Such rules, whilst not a query language per se, can be used for the purpose of querying. For this purpose queries have to be stored as rules in a dedicated file, which is used in conjunction with the data. The CWM filter command allows to automatically select the data that is generated by rules. Even though N3 fulfills the orthogonality, closure and safety property, using N3 as a query language is cumbersome.

N3 is supported by two freely available systems, i.e. Euler [Roo, 2002] and CWM [Berners-Lee, 2000]. None of these systems do automatically adhere to the RDF semantics. The semantics has to be provided by custom rules. For our comparison, we worked with CWM in the version of March 21, 2004.

Versa

The Versa language was developed by Mike Olson and Uche Ogbuji[10]. Versa takes an interesting approach in that the main building block of the language is a list of RDF resources. RDF triples play a role in the so-called traversal operations, which have the form `ListExpr - ListExpr -> BoolExpr`. These expressions return a list of all objects of matching triples. For instance, the traversal expression `all() - rdfs:label -> *` would return a list containing all labels. Within a traversal expression, we can alternatively select the subjects as well by placing a vertical bar at the beginning of the arrow symbol. Thus, `all() |- rdfs:label -> eq("foo")` would yield all resources having the label "foo". The fact that a traversal expression is again a list expression, allows us to nest expressions in order to create more complex queries. Besides lists, Versa defines the types boolean, number, string, Resource, and set along with several conversion functions. Other functions, such as sort, return one of the respective types and can therefore be included in the queries as well.

The given data structures and expression tree make it hard to project several values at once. Versa uses the distribute operator to work around this limitation. It creates a list of lists, which allows selecting several properties of a given list of resources.

Versa offers some support for rules since it allows traversing predicates transitively. Custom built-ins, views, multiple models, and data manipulation are not implemented. However, Versa fulfills the orthogonality and safety criteria.

The Versa language is supported by 4Suite, which is a set of XML and RDF tools[11]. We used 4Suite version 1.0a3 for Windows from July 4, 2003 along with Python 2.3.

[10] http://uche.ogbuji.net/tech/rdf/versa/

[11] `http://www.4suite.org`

3.6 RDF Querying Use Cases

In this section we present use cases for the querying of RDF data and evaluate how the six query languages support them. In the following tables, "-" indicates no support, "•" full support and "○" partial support.

3.6.1 Sample Data

For our comparison, we have used a sample data set[12]. It describes a simple scenario of the computer science research domain, modelling persons, publications and a small topic hierarchy. The data set covers the main features of the RDF data model. It includes a class hierarchy with multiple inheritance, data types, resources with multiple instantiations, reification, collections of resources, etc. These variety of features are exploited in the following use cases.

3.6.2 Use Case Graph

Due to RDF's graph-based nature, a central feature of most query languages is the support for graph matching.

Path Expressions

The central feature used to achieve this matching of graphs is a so-called path expression, which is typically used to traverse a graph. A path expression can be decomposed into several joins and is often implemented by joins. It comes at no surprise that path expressions are offered - in various syntactic forms - by all RDF query languages.

Return the names of the authors of publication X

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Path	•	•	•	•	•	•

Optional Path Expressions

The RDF graph represents a semi-structured data model. Its weak structure allows to represent irregular and incomplete information. Therefore RDF query languages should provide means to deal with irregularities and incomplete information. A particular irregularity, which has to be accounted for in the following query, is that a given value may or may not be present.

What are the name and, if known, the e-mail of the authors of all available publications ?

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Optional Path	-	-	•	•	-	○

[12] Available at `http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/sample.rdf`

Unfortunately, only two languages - namely Versa and SeRQL - provide built-in means for dealing with incomplete information. For example, the SeRQL language provides so-called optional path expressions (denoted by square brackets) to match paths whose presence is irregular:

```
SELECT PersonName, Email
FROM {X} ns3:author {} rdfs:member {p} ns3:name {PersonName};
                                       [ns3:email {Email}]
USING NAMESPACE
   ns3 = <...>
```

Usually, such optional path expressions can be simulated, if a language provides set union and negation. A correct answer is provided by unifying the results of two select-queries, where the first argument of the union retrieves all persons with an e-mail address, the second those without an e-mail address. Consequently, RQL gets partial credit since these operations are supported.

```
( select PersonName, Email
  from {X} s:author {y}. rdfs:member {z}.
  s:name {PersonName}, {z} s:email {Email}
) union (
  select PersonName, NULL
  from {X} s:author {y}. rdfs:member {z}.
  s:name {PersonName}
  where not ( z in select X from {X}
  s:email {e} ) )
using namespace
  s = ... , rdfs = ...
```

Another workaround is possible in rule languages like N3 and Triple by defining a rule, which infers a dummy value in absence of other data. In our example, an optional path would carry the dummy email address. However, this workaround might create undesired results if the absence of values is checked in other parts of the query.

Versa's distribute operator allows formulating the query by converting the list of authors into a list of lists of (optional) attributes.

```
distribute(((@"../versa-sample.rdf#Paper" - s:author -> *)
 - properties(.) -> *), ".- s:name -> *", ".- s:email -> *")
```

3.6.3 Use Case Relational

RDF is frequently used to model relational structures. In fact, n-ary tables such as found in the the relational data model can easily be encoded in RDF triple.

Basic algebraic operations

In the relational data model several basic algebraic operations are considered, i.e. (i) selection, (ii) projection, (iii) cartesian product, (iv) set difference and (v) set union. These operations can be combined to express other operations such as set intersection, several forms of joins, etc. The importance of these operations is accounted by the definition of relational completeness. In the relational world, a given query languages is known to be relationally complete, if it supports the full range of basic algebraic operations mentioned above.

The three basic algebraic operations selection, projection, product are supported by all languages and have been used in the path expression query of the previous use case *Graph*. We therefore concentrate on the other two basic operations mentioned above, i.e. union and difference, in this section.

Union

As we have seen in the previous section, union is provided by RQL. Versa contains an explicit union operator as well. N3 and Triple can simulate union with rules.

Return the labels of all topics and (union) the titles of all publications

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Union	-	●	-	●	●	●

Difference

Difference is a special form of negation:

Return the labels of all topics that are not titles of publications

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Difference	-	-	-	○	-	●

While difference is described in the Versa documentation (but not implemented) RQL also provides an implementation of this algebraic operator.

The following RQL query provides the correct answer:

```
( select title
  from s:Topic{T}. rdfs:label {title}
) minus (
  select title
  from s:Publication{P}. s:title {title} )
using namespace
  s = ... , rdfs = ...
```

Quantification

An existential predicate over a set of resources is satisfied if at least one of the values satisfies the predicate. Analogously, a universal predicate is satisfied if all the values satisfy the predicate. As any selection predicate implicitly has existential bindings, we here consider universal quantification.

Return the persons who are authors of all publications.

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Quantification	-	-	-	-	-	●

RQL is the only language providing the universal quantification needed to answer this query:

```
SELECT person
FROM s:Person{person}
WHERE FORALL z IN
    (SELECT x FROM s:Publication{x} )
  SUCH THAT EXISTS p IN
    (SELECT Y FROM {z} s:author {}. rdfs:member {y})
  SUCH THAT person = p
USING NAMESPACE s = ..., rdfs = ...
```

3.6.4 Use Case Aggregation and Grouping

Aggregate functions compute a scalar value from a multi-set of values. These functions are regularly needed to count a number of values. For example, they are needed to identify the minimum or maximum of a set of values. Grouping additionally allows aggregates to be computed on groups of values.

Aggregation

A special case of aggregation tested in the following query is a simple count of the number of elements in a set:

Count the number of authors of a publication.

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Counting	-	-	-	•	•	•

Counting is supported by N3, Versa and RQL. The following N3 rule gives the appropriate answer:

```
{?y.sam:author math:memberCount ?result .} =>
{:Query :Result ?result}.
```

Grouping

None of the compared query languages allows to group values, such as provided with the SQL GROUP BY clause[13].

3.6.5 Use Case Recursion

Recursive queries often appear in information systems, typically if the underlying relationship is transitive in nature. Note that the RDF query engine must handle *schema* recursion, i.e. the transitivity of the subClassOf relation. The scope of this use case is *data* recursion introduced by the application domain. In the sample datasetour comparison, topics are defined along with their subtopics, where the subtopic property is transitive. This must be expressed in the query.

Return all subtopics of topic "Information Systems", recursively.

[13]The RQL query language allows the computation of global aggregate values such as required for a query such as selecting the publication with the maximum number of authors.

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Recursion	-	●	-	●	●	-

RQL and SeRQL, by means of their explicit commitment to RDFS semantics, allow recursion along the properties rdfs:subClassOf and rdfs:subPropertyOf, but lack the means to express recursion along arbitrary properties.

Triple (and N3), being rule-based systems, naturally can support the required recursion through the definition of auxiliary rules:

```
FORALL O,P,V  O[acm:SubTopic->V] <-
  EXISTS W  (O[acm:SubTopic->W] AND W[acm:SubTopic->V])@default:ln.
FORALL Y <-
    ('...#ACMTopic/':Information_Systems[acm:SubTopic->Y])@default:ln.
```

Versa does not support general recursion, but provides a keyword "traverse", which effects a transitive interpretation of a specified property. This suffices to answer our recursive query:

```
traverse(@"...#ACMTopic/Information_Systems",
    acm:SubTopic, vtrav:forward, vtrav:transitive )
```

3.6.6 Use Case Reification

Reification is a unique feature of RDF. It adds a meta-layer to the graph and allows treating RDF statements as resources themselves, such that statements can be made about statements. In the sample data reification is used to state who entered publication data. Hence, the following query is of interest:

Return the person who has classified the publication X.

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Reification	○	○	●	○	-	○

SeRQL and Triple support reification with a special syntax. In SeRQL, a path expression representing a single statement can be written between the curly brackets of a node:

```
select Person
from {{X} s:isAbout {}} dc:creator {Person}
using namespace s = <...>
```

In Triple statements can be reified by placing them in angle brackets and using this within another statement: `FORALL V,W,X,Y,Z <- ( V[W-><X[Y->Z]>] ).` However, we were only able to use this feature in the native F-Logic syntax, since the reified statements in the RDF sample data were not parsed correctly.

While N3 cannot syntactically represent RDF reification, RDQL, RQL and Versa treat reified statements as nodes in the graph, which can be addressed through normal path expressions by relying on the RDF normalization of reification. This allows treating the reification use case like any other query. We show the Versa example below:

```
QUERY=(all() |- rdf:predicate -> s:isAbout) - dc:creator -> *
```

3.6.7 Use Case Collections and Containers

RDF allows to define groups of entities using collections (a closed group of entities) and containers, namely `Bag`, `Sequence` and `Container`, which provide an intended meaning of the elements in the container. A query language should be able to retrieve the individual and all elements of these collections and containers, along with order information, if applicable, as in the following query:

Return the last author of Publication X

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Sequences	○	○	○	○	○	○

Although none of the query languages provide explicit support for the processing of containers (as known for example from the processing of sequences in XQuery), in all query languages it is possible to query for a particular element in a container with the help of the special predicate <rdf:_n>, which allows to address the nth element in a container. However, this approach only allows to retrieve the last element of a container if its size is known before.

None of the query languages provide explicit support for ordering or sorting of elements, except for Versa which features a special sort operator. RQL does have specific operators for retrieval of container elements according to its specification, but this feature is not implemented in the current engine.

3.6.8 Use Case Namespaces

Namespaces are an integral part of any query language for web-based data. The various examples presented so far showed how the languages allow introducing namespace abbreviations in order to keep the queries concise. This use case evaluates which operations are possible on the namespaces themselves. Given a set of resources, it might be interesting to query all values of properties from a certain namespace or a namespace with a certain pattern. The following query addresses this issue. Pattern matching on namespaces is particularly useful for versioned RDF data, as many versioning schemes rely on the namespace to encode version information.

Return all resources whose namespace starts with "http://www.aifb.uni-karlsruhe.de/".

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Namespace	○	-	●	-	●	●

SeRQL, RQL and N3 allow for pattern matching predicates on URIs in the same manner as for literals, which allows to realize the query as shown in the following for N3:

```
{?a ?b ?c. ?a log:rawUri ?d.
 ?d string:startsWith "http://www.aifb.uni-karlsruhe.de/" } =>
{:Query :Result ?d}.
```

For RDQL, the string match operator is defined in the grammar, however the implementation is incomplete. Versa has a contains operator, which apparently only works for string literals, not URIs.

3.6.9 Use Case Language

RDF allows to use XML-style language tagging. The XML tag enclosing an RDF literal can optionally carry an xml:lang attribute. The respective value identified the language used in the text of the literal. Possible values include en for english or de for german. This use case examines, whether the various languages support this RDF feature.

Return the German label of the topic whose English label is "Database Management"

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Language	-	-	•	-	-	-

Out of the compared languages, SeRQL is the only one that has explicit support to query language specific information. SeRQL provides a special function to retrieve the language information from a literal:

```
select deLabel
from {} rdfs:label {deLabel, enLabel}
where lang(deLabel) = "de" and lang(enLabel) = "en" and
      label(enLabel) = "Database Management"
```

3.6.10 Use Case Literals and Datatypes

Literals are used to identify values such as numbers and dates by means of a lexical representation. In addition to plain literals, RDF supports the type system of XML Schema to create typed literals. An RDF query language should support the XML Schema datatypes. A datatype consists of a lexical space, a value space and lexical to value mapping. This distinction should also be supported by an RDF query language. The sample data for example contains a typed integer literal to represent the page number of a publication, with the following two queries we will query both the lexical space and the value space:

Return all publications where the page number is the lexical value '08'

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Lexical Space	•	•	•	•	•	•

```
select Pub
from {Pub} rdf:type {ns3:Publication};
           ns3:pages {X}
where X like "08"
using namespace
   ns3 = <http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/sample.rdf#>
```

Return all publications where the page number is the integer value 8

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Value Space	○	-	●	-	-	●

```
select Pub
from {Pub} rdf:type {ns3:Publication};
           ns3:pages {X}
where X = "8"^^xsd:integer
using namespace
   ns3 = <http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/sample.rdf#>
```

All query languages are able to query the lexical space, but most query languages have no or only preliminary support for datatypes and do not support the distinction between lexical and value space. RQL does have full datatyping support, RDQL and SeRQL provide support for datatypes using a special syntax to indicate the datatype. However, the RDQL query did not work correctly in the tested implementation.

3.6.11 Use Case Entailment

RDF Schema vocabulary supports the entailment of implicit information. Two typical use cases in the context of RDF are:

- Subsumptions between classes and properties that are not explicitly stated in the RDF Schema,
- Classification of resources: For a resource having a property for which we know its domain – or analogously the range – is restricted to a certain class, we can infer the resource to be an instance of that class.

However, only a few of the RDF query languages actually interpret RDF Schema information.

With the following query we evaluate the support of the query languages for RDF Schema entailment. The query is expected to return not only the resources for which the class membership is provided explicitly, but also those whose class membership can be inferred based on the entailment rules. Obviously, several other queries would be necessary to test the full RDF-S support. Due to space limitations, we restrict ourselves to the following query:

Return all instances of that are members of the class Publication

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Entailment	○	○	●	-	○	●

3.6.12 Discussion

Regarding the support for RDF-S entailment, the query languages take different approaches: RQL and SeRQL support entailment natively and even allow to distinguish between subclasses and direct subclasses. RDQL leaves entailment completely up to the implementation of the query engine, it is thus not part of the semantics of RDQL. Nevertheless, it gets partial credit, since Jena provides an optional mechanism for attaching

an RDF-S reasoner such that the query processor takes advantage of it. N3 and Triple require an axiomatization of the RDF-S semantics, i.e. a set of rules. Versa provides no support.

The following sample shows how the query is realized in RQL:

```
select publications
from ns3:Publication{publications}
using namespace
    ns3 = http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/sample.rdf#
```

3.7 Summary and Wish List

In table 3.1 we summarize our findings by again listing each use case and which languages completely or partially support it.

Query	RDQL	Triple	SeRQL	Versa	N3	RQL
Path	●	●	●	●	●	●
Optional Path	-	-	●	●	-	○
Union	-	●	-	●	●	●
Difference	-	-	-	○	-	●
Quantification	-	-	-	-	-	●
Counting	-	-	-	●	●	●
Recursion	-	●	-	●	●	-
Reification	○	○	●	○	-	○
Sequences	○	○	○	○	○	○
Namespace	○	-	●	-	●	●
Language	-	-	●	-	-	-
Lexical Space	●	●	●	●	●	●
Value Space	○	-	●	-	-	●
Entailment	○	○	●	-	○	●

Table 3.1: Summary of feature support

The table shows that none of the tested query languages supports all the formulated use cases. In the previous sections we have seen that the currently available query languages for RDF support a wide variety of operations. However, several important features are not well supported, or even not supported at all.

- *Grouping and Aggregation*
  Many of the existing proposals support little functionality for grouping and aggregation. Functions such as min, max, average and count provide important tools for analysing data. In table 3.1 this is underlined by the relatively low score on the use case for counting.

- *Sorting*
  Perhaps surprisingly, except for Versa, no language is capable to do sorting and

ordering on the output. Related to this seems to be that many query languages do not support handling of ordered collections.

- *Optional matching*
  Due to the semi-structured nature of RDF, support for optional matches is crucial in any RDF query language and should be supported with a dedicated syntax. Table 3.1 shows that only a few query languages currently support this.

- *Adequacy*
  Overall, the languages' support for RDF-specific features like containers, collections, XML Schema datatypes, language tags, and reification is quite poor. Since these are features of the data model, the degree of adequacy among the languages is low. For instance, it would be desirable for a query language to support operators on XML Schema datatypes, as defined in [Malhotra et al., 2003], as built-ins.

3.8 Conclusion

In this chapter, we have outlined several general requirements for querying semi-structured data, and have explicated and extended these requirements for the specific case of querying RDF. We have demonstrated a number of RDF querying approaches and highlighted their features and shortcomings in terms of a use case-oriented testing framework.

We believe that defining a suitable RDF query language should be a top priority in terms of standardization. Querying is a fundamental functionality required in almost any Semantic Web application. Judging from the impact of SQL to the database community, standardization will definitely help the adoption of RDF query engines, make the development of applications a lot easier, and will thus help the Semantic Web in general.

We have evaluated six query language proposals with quite different approaches, goals, and philosophies. Consequently, it is hard to compare the different proposals and come up with a ranking. From our analysis, we identify a small set of key criteria, which differ vastly between the languages.

A key distinction is the support for RDF Schema semantics. Languages like N3 and Triple do not make a strict distinction between queries and rules. Thus, a logic program representing the desired semantics, in this case RDF-S, can optionally supplement a query. SeRQL and RQL support RDF-S semantics internally. Versa takes a pragmatic approach by supporting the transitive closure as a special operator. While this is not very flexible, it solves most of the problems like traversing a concept hierarchy. RDQL's point of view is that entailment should be completely up to the RDF repository.

Orthogonality is a desirable feature, since it allows combining a set of simple operators into powerful constructs. Out of the six candidates, RQL, SeRQL, N3, and Versa support this property. Versa uses sets of resources as the basic data structure, whereas RQL, N3 and SeRQL operate on graphs. Triple can mimic orthogonality via rules, whereas RDQL does not support it.

Furthermore, we consider the extent to which the various use cases are supported. Obviously, one would have to distinguish between features like the support for recursive

queries that fundamentally cannot be expressed in a language and a feature that simply has not been implemented in the respective system like a simple string match operator. However, since this distinction is often hard to make, we simply add up the queries that could be expressed. If the query could be formulated with a workaround, we count half a point. Using this metric, RQL and SeRQL appear to be the most complete languages, covering 10.5 and 8.5 out of 14 use case queries. Versa and N3 follow with 7.5 and 7. Triple and RDQL were able to answer the least queries and got 5.5 and 4.5 points.

Finally, we consider the readability and usability of a language. Obviously, this depends on personal taste. Syntactically, RQL, RDQL, and SeRQL are similar due to their SQL / OQL heritage. Triple and N3 share the rules character. The Triple syntax allows for some nice syntactic variants. Versa's style is quite different, since a query directly exposes the operator tree.

Chapter 4

SeRQL: A Second Generation RDF Query Language

In this chapter, we present the SeRQL query language in detail. We outline the design decisions behind the language, present its syntax and formal semantics in some detail, and couple the design decisions to the query language requirements outlined in chapter 3.

4.1 Introduction

RDF Query Language proposals are numerous. However, the most prominent proposals are query languages that were conceived as first generation tryouts of RDF querying, with little or no RDF-specific implementation and use experience to guide design, and based on an ever-changing set of syntactical and semantic specifications. Now that the revised RDF specifications have reached the status of Recommendation, it is the right time to reevaluate the design of the current set of query languages.

In this chapter, we introduce the new RDF query language SeRQL. SeRQL was designed using experiences gained from design and implementation of other query languages and from feedback received from users and developers of these query languages and the systems in which they were implemented, such as Sesame [Broekstra et al., 2002]. SeRQL's aim is to reconcile ideas from existing proposals (most prominently RQL [Alexaki et al., 2000], RDQL [Seaborne, 2004], N-Triples [Grant and Beckett, 2003] and N3 [Berners-Lee, 1998a]) into a proposal that satisfies a list of key requirements, and thus offer an RDF query language that is powerful, easy to use and adresses practical problems one encounters when querying RDF.

This chapter is organized as follows: in section 4.2, we present a list of principles and requirements to which an RDF query language should conform. In section 4.3, we introduce the syntax and design of the SeRQL query language. In section 4.3.4, we briefly summarize how SeRQL addresses the list of requirements. In section 4.4, we define a formal interpretation of SeRQL. In section 4.5, we discuss some real-life examples of the application of SeRQL. Finally we present our conclusions in section 4.7.

4.2 Query Language requirements

In this section, we will look at the general query language requirements that were identified in chapter 3, section 3.5.

In [Reggiori and Seaborne, 2002], Alberto Reggiori and Andy Seaborne have collected a number of use cases and examples for RDF queries. From this report, we can distill several general requirements for RDF queries, most notably expressivity requirements. Apart from these requirements, several general principles for query languages can be taken into account, such as compositionality, and data model awareness. From these sources and our experience in implementing and using first generation RDF query languages such as RQL and RDQL, we have composed a list of key requirements for RDF querying. In the next sections, we briefly discuss these requirements and show how SeRQL aims to fulfill them.

4.2.1 Expressiveness and Adequacy

In [Abiteboul et al., 1999] it is noted that the notion of expressiveness is rather ill-defined for semistructured data models in general (due to the imprecision of the notion of "semistructured"). However, we can write down an informal list of the kinds of operations that a query language should express. Expressiveness requirements that have come up often in dialogue with RDF developers (see also [Reggiori and Seaborne, 2002]) and users of the Sesame system[1] include:

1. A convenient yet powerful path expression syntax for traversing the RDF graph.
2. Functionality for traversing the class/property hierarchy.
3. Functionality for querying reified statements.
4. Value comparison and datatype support.
5. Functionality to deal with *optional* values; properties which may or may not be present in the data for a particular resource.

Of course, this list is far from exhaustive, but these requirements illustrate practical applications of an RDF query language.

Additionally, a query language is called *adequate* when it has specific support for every primitive in the model that it intends to query (see section 4.2). In terms of RDF, this means that, additional to the functions mentioned above, it should have support for querying literal facets (language and datatype), RDF containers and lists, and be aware of the semantics of RDF Schema primitives.

[1] See a.o. the Sesame developers web forum at `http://www.openrdf.org/forum`

4.2.2 Schema awareness

From the point of view of adequacy, query languages should be schema aware. When structure is defined or inferred, a query language should be capable of exploiting the schema for type checking, optimization, and entailment.

This requirement is closely tied with the requirement for formal semantics (section 4.2.5) and for expressiveness and adequacy. In the case of RDF, it means that the query language should be aware of the semantics of RDF and RDF Schema as they are specified by the RDF model theory.

4.2.3 Program manipulation

It is important that the query language is simple enough to allow program-generated queries. This means that it is often preferable to use a query language syntax that is easy to parse and decompose, rather than try and make it as 'user-friendly' as possible (at the risk of making it ambiguous and thus harder to process).

Nevertheless, there is a balance to be obtained here: a query language that is unintelligable to humans will not find acceptance, no matter how well it can be processed automatically.

Considerations to take into account with respect to this requirement include such things as simplicity of structure and avoiding redundancy, while keeping a balance with convenience and readability.

4.2.4 Compositionality

A query language is *compositional*, or *closed*, when the output of a query can be used as the input of another query. This is useful in situations where one wants to decompose large queries into smaller ones, or when one wants to execute several queries in series, using the output of the first as the input for the second, etc. A query language with this property will also be able to facilitate view definitions.

In the case of an RDF query language, compositionality obviously means that the result of a query should be representable as an RDF graph. The effect of this is that the query language functions as a *transformation language* on RDF graphs.

4.2.5 Semantics

Precise formal semantics of a query language are important, because without these query transformations and optimizations are virtually impossible. Moreover, formal descriptions avoid ambiguity and thus help prevent different implementations of the same language interpreting queries differently.

SeRQL is provided a formal semantics by specifying a mapping between query constructs and the RDF model theory, as specified in [Hayes, 2004].

4.3 The Syntax of SeRQL

SeRQL (Sesame RDF Query Language, pronounced 'circle') is a new RDF/RDFS query language that was developed to address practical requirements from the Sesame user community[2] that were not sufficiently met by other query languages. SeRQL combines the best features of other languages and adds some of its own.

In the rest of this section, we will give an overview of the basic syntax of SeRQL. The overview of the SeRQL language that is given here is not intended to be complete. A full manual for writing SeRQL queries that covers the complete language is available on the Web [Broekstra and Kampman, 2003].

4.3.1 URIs, Literals and Variables

URIs and literal values are the basic building blocks of RDF. In SeRQL, URIs are denoted using a syntax derived from N-Triples and N3 notation. Full URIs must be surrounded with < and >, as follows: `<http://www.openrdf.org/index.jsp>`. As URIs tend to be long strings with the first part being shared by several of them (i.e. the namespace), SeRQL allows one to use abbreviated URIs by defining prefixes for these namespaces. An example abbreviated URI is: `sesame:index.jsp`.

RDF literals consist of one or two parts: a label and optionally a language tag, or a datatype. The notation of literals in SeRQL has been modelled after their notation in N-Triples; literals start with the label, which is surrounded by double quotes, optionally followed by a language tag with a `@` prefix, or by a datatype URI with a `^^` prefix.

Example literals are:

- `"foo"`
- `"foo"@en`
- `"foo"^^<http://some/datatype>`

In SeRQL, variables are identified by names. These names must start with a letter or an underscore ('_') and can be followed by zero or more letters, numbers, underscores, dashes ('-') or dots ('.'). Variable names are case-sensitive. SeRQL keywords are not allowed to be used as variable names.

4.3.2 Path Expressions

SeRQL path expressions are expressions that match specific paths through an RDF graph. Most current RDF query languages allow one to define path expressions of length 1, which can be used to find (combinations of) triples in an RDF graph. SeRQL, like RQL, allows one to define path expressions of arbitrary length.

SeRQL uses a path expression syntax that is similar to the syntax used in RQL, and is based on the graph nature of RDF: the path is expressed as a collection of nodes and edges, where each node is denoted by surrounding curly brackets.

[2] See `http://www.openrdf.org/`

```
{node} edge {node} edge {node}
```

As an example, suppose we want to query an RDF graph for persons that work for an IT Company. A path expression to express this could look like:

```
{Person} foo:worksFor {Company} rdf:type {foo:ITCompany}
```

Notice that resource URIs and variables are intermixed to provide a template which is matched against the RDF graph.

Multiple path expressions can be comma-seperated. For example, we can split up the above path expression into two simpler ones:

```
{Person} foo:worksFor {Company},
{Company} rdf:type {foo:ITCompany}
```

SeRQL allows variable repetition to express implicit joins.

Extended Path Expressions

As we have just seen, SeRQL has a convenient syntax for basic path expressions, which can be composed into path expressions of arbitrary length. Every path in an RDF graph can be expressed using these basic path expressions. However, several extended constructions are supported to allow for more convenient expressions of paths.

In situations where one wants to query for two or more triples with identical subject and predicate, the subject and predicate do not have to be repeated. Instead, a *multi-value node* can be used, for example to express three distinct values of a particular subject and property:

```
{subj1} pred1 {obj1, obj2, obj3}
```

This path expression is equivalent to:

```
{subj1} pred1 {obj1},
{subj1} pred1 {obj2},
{subj1} pred1 {obj3}
```

SeRQL also introduces the notion of *branched path expressions*. This is a construction that is useful when multiple properties that emenate from a single node are queried. The semi-column is used to denote a branch:

```
{subj1} pred1 {obj1};
        pred2 {obj2}
```

which is equivalent to:

```
{subj1} pred1 {obj1},
{subj1} pred2 {obj2}
```

A slightly more complicated example of a branched path expression:

```
{subj1} pred {} pred1 {obj1};
                pred2 {obj2} pred3 {obj3}
```

The empty curly brackets represent a node in the graph that we have no interest in save as a connection point between `pred` and `pred1`. Thus, the node is not assigned a variable.

Reification

RDF allows for a syntactic construction known as *reification*, where the subject or object of a statement is itself a statement. Since it is a syntactic construction it can be expressed using basic path expression syntax, as follows:

```
{statement1} rdf:type {rdf:Statement},
{statement1} rdf:subject {subj1},
{statement1} rdf:predicate {pred1},
{statement1} rdf:object {obj1},
{statement1} pred2 {obj2}
```

However, this is a cumbersome way of dealing with reification. SeRQL introduces a shorthand notation for reified statements that allows one to treat reified statements as actual objects instead of the complex syntactic structure shown above. In this notation, the above reified statement would become:

```
{{subj1} pred1 {obj1}} pred2 {obj2}
```

Class and Property Hierarchies

In the previous section we have shown how the RDF graph can be navigated through path expressions. The same principle can be applied to navigation of class and property hierarchies, since these are, of course, also graphs. For example, to retrieve the subclasses of a particular class `my:class1`:

```
{subclass} rdfs:subClassOf {my:class1}
```

Or, to retrieve all instances of class `my:class1`:

```
{instance} rdf:type {my:class1}
```

However, an RDFS class/property hierarchy encapsulates notions such as inheritance, which must be taken into account. Therefore, SeRQL applies the RDF Schema semantics when this is required. In the case of the property `rdfs:subClassOf`, for example, SeRQL will not only return all explicitly asserted subClass relations, but also the ones that are entailed according to the model theory.

Additionally, SeRQL supports a numer of *built-ins* for expressing queries about the class hierarchy. These built-ins are 'virtual' properties, that is, they are used as normal properties in path expressions, but this property is not expected to actually occur in the RDF graph. Instead, the meaning of the property is pre-defined in terms of other properties.

SeRQL supports three built-ins: `serql:directSubClassOf`, `serql:directSubPropertyOf` and `serql:directType`. We give the definition of `serql:directSubClassOf` here:

Definition 1 *A is a* `serql:directSubClassOf` *B if A and B are not equal and there is no class $C \neq A \neq B$ such that C is a subclass of B and a superclass of A.*

It is important to note that these built-ins are not merely syntax shortcuts, but actually provide additional expressivity: the notion of direct subclass/property/instance can not be expressed using normal path expressions and boolean constraints only (it would require set quantification operations).

Optional Matches

The path expressions and boolean constraints introduced sofar provide the means to specify a template that *must* match the RDF graph in order to return results. However, since the RDF data model is by its very nature weakly structured (or *semi-structured*), it is important that an RDF query language has the means to deal with data that does not strictly conform to a schema.

In contrast to query languages for strongly structured data models, such as SQL, RDF query languages must be able to cope with the possibility that a given value may or may not be present. In SeRQL, such values are called *optional matches*. The query language facilitates optional matches by introducing a square-bracket notation that encloses the optional part of a given path expression.

Consider an RDF graph that contains information about people that have names, ages, and optionally e-mail addresses, that is, for some people the e-mail address is known, but for others, it is not. This is a situation that is likely to be very common in RDF data. A logical query on this data is a query that yields all names, ages and, when available, e-mail addresses of people. A path expression to retrieve these values would look like this:

```
{Person} person:name {Name};
         person:age  {Age};
         person:email {EmailAddress}
```

However, using normal path expressions like in the query above, people without e-mail address will not be matched by the template specified by this path expression, and their names and ages will not be returned by the query. With optional path expressions, one can indicate that a specific (part of a) path expression is optional. This is done using square brackets:

```
{Person} person:name {Name};
         person:age  {Age};
        [person:email {EmailAddress}]
```

In contrast to the first path expression, this expression will also match with people without an e-mail address. For these people, the variable `EmailAddress` will not be assigned a value.

Optional path expressions can also be nested. This is useful in situations where the existence of a specific path is dependent on the existence of another path. For example, the following path expression queries for the titles of all known documents and, if the author of the document is known, the name of the author (if it is known) and his e-mail address (if it is known):

```
{Document} foo:title {Title};
          [foo:author {Author} [foo:name {Name}];
                               [foo:email {Email}]]
```

There are a few restrictions on the use of variables in optional path expressions. Most importantly, two optional path expressions that are in parallel to each other (that is, one is not nested within the other) may only have a shared variable if that variable is constrained to a value *outside* either of the optional expressions.

For example, the optional path expressions `foo:name {Name}` and `foo:email {Email}` share the subject-variable `Author`. This is allowed only because this variable is constrained by the path expression `foo:author {Author}`, that is, outside the two parallel optional path expressions.

The reason for this restriction becomes apparent when we consider the following example query[3]:

```
select *
from [{<x>} <p> {a}], [{<x>} <q> {a}]
```

In this example, the variable `a` is shared between two parallel optional expressions, but it is not otherwise constrained. Now, we further assume that the RDF graph contains the following two RDF statements: `<x> <p> <y>` and `<x> <q> <z>`.

In this setting, the variable `a` can be unified with the value `<y>` or with `<z>`, but not both at the same time. The query causes an ambiguity: depending on the order in which the optional expressions are evaluated, the variable gets assigned a different value. Since such order dependency is an undesirable feature in a declarative language, we restrict the language to prevent this.

4.3.3 Filters and operators

In the preceding sections we have introduced several syntax components of SeRQL. Full queries are built using these components, and using an RQL-style `select-from-where` (or `construct-from-where`) filter. Both filters additionally support a `using namespace` clause. Queries specified using the `select-from-where` filter return a table of values, or a set of variable-value bindings. Queries using the `construct-from-where` filter return an RDF graph, which can be a subgraph of the graph being queried, or a graph containing information that is derived from it.

The select and construct clauses

The first clause (i.e. `select` or `construct`) determines what is done with the results that are found. In a `select` clause, one can specify which variable values should be returned and in what order, by means of a comma-seperated list of variables. Optionally, it is possible to use a `*` instead of such a list to indicate that all variables that are used should be returned, in the order in which they appear in the query.

For example, the following query retrieves all classes:

```
select C
from  {C} rdf:type {rdfs:Class}
```

In a `construct` clause, one can specify which triples should be returned. Construct queries, in their simplest form, simply return the subgraph that is matched by the template specified in the `from` and `where` clauses. The result is returned as the set of triples that make up the subgraph. For example:

[3]Example by Andy Seaborne and Jeremy Carroll, see `http://lists.w3.org/Archives/Public/www-rdf-interest/2003Nov/0076.html`

```
construct *
from {SUB} rdfs:subClassOf {SUPER}
```

This query extracts all triples with a `rdfs:subClassOf` predicate from an RDF graph.

Construct queries can also be used to do *graph transformations* or to specify simple rules. Graph transformation is a powerful tool in application scenarios where mappings between different vocabularies need to be defined. As a simple example, consider the following construct query:

```
construct {Parent} my:hasChild {Child}
from {Child} foo:hasParent {Parent}
```

This query can be interpreted as a rule that specifies the inverse of the `foo:hasParent` relation. More generally, it specifies a graph transformation: the original graph may not know the `foo:hasChild` relation, but the result of the query is a graph that contains `foo:hasChild` relations between parents and children. The construct clause allows the introduction of new vocabulary, so this query will succeed even if the relation `my:hasChild` is not present in the original RDF graph.

The from clause

The `from` clause always contains path expressions. It defines the paths in an RDF graph that are relevant to the query and binds variables to values.

The where clause

Finally, the `where` clause is optional and can contain additional boolean constraints on the values in the path expressions. These are constraints on the nodes and edges of the paths, which cannot always be expressed in the path expressions themselves.

SeRQL contains a set of operators for comparing variables and values that can be used as boolean constraints, including (sub)string comparison, datatyped numerical comparison and a number of boolean functions.

As an example, the following query uses a datatyped comparison to select countries with a population of less than 1 million.

```
SELECT Country
FROM {Country} foo:population {Population}
WHERE Population < "1000000"^^xsd:positiveInteger
```

SeRQL will try to cast both arguments of the operator to the specified datatype. If no datatype was specified in the query itself, the behaviour of the comparison would depend on the bound value for `Population`: for non-datatyped literals it would perform a lexical comparison, for datatyped literals it would perform the appropriate numerical comparison. For a full overview of the available operators and functions, see the SeRQL user manual [Broekstra and Kampman, 2003].

The using namespace clause

The `using namespace` clause is also optional and it can contain namespace declarations; these are the mappings from prefixes to namespaces for use in combination with abbreviated URIs (see section 4.3.1).

4.3.4 Requirements revisited

In chapter 3 and section 4.2, we identified a list of requirements that should hold for RDF query languages. In this section, we will briefly revisit these requirements and show how each requirement is fulfilled by SeRQL.

- **Expressive power and Adequacy**
  In the preceeding sections, we have illustrated SeRQL's expressivity and adequacy in querying RDF in some detail. Specifically, we have shown how SeRQL handles *optional matches*, *reification* and *schema queries*. SeRQL's path expression syntax has been shown to be very powerful, and we have briefly touched upon datatyping in section 4.3.3.

- **Schema awareness**
  In section 4.3.2, we have shown how SeRQL handles schema interpretation by retrieving not just explicitly asserted statements, but also those statements that are implied by the RDF Schema semantics.

- **Compositionality**
  In section 4.3.3, we have shown how SeRQL queries can be used for transformation or composition using the `construct` clause.

- **Program manipulation**
  As has been shown in the previous sections, SeRQL has a syntax that is designed to be unambiguous and structured. These properties make it ideally suited to queries being formulated and analysed through programmatic means.

- **Formal semantics**
  SeRQL is grounded in the RDF Model Theory. In section 4.4, we present a formal interpretation of SeRQL queries.

If we look at SeRQL in terms of the general language properties identified in chapter 3, section 3.2, we see the following:

- SeRQL's **expressiveness** is high but not complete, as it, for example, does not currently support relational algebraic operations (such as union, intersection and difference).

- SeRQL is **closed** as it supports query result sets that are RDF graphs.

- SeRQL is almost but **not completely adequate**: it supports most concepts in the RDF data model and is schema-aware, but it currently has no explicit support for handling containers.

- SeRQL is **orthogonal**: every operation is generalized for any RDF graph and is independent of context.
- SeRQL is **safe**: every syntactically correct query returns a finite set of results (given a finite data set).

4.4 Formal Interpretation of SeRQL

In this section, we introduce a formal grounding for SeRQL queries.

4.4.1 Mapping Basic Path Expressions to Sets

The RDF Semantics W3C specification [Hayes, 2004] specifies a model theoretical semantics for RDF and RDF Schema. In this section, we will use this model theory to specify a formal interpretation of SeRQL query constructs. Without repeating the entire model theory, we summarize a couple of its notions for reference:

- The sets IR, IP, IC are sets of resources, properties, and classes, respectively. LV is a distinguished subset of IR and is defined as the set of literals.
- $IEXT$ is defined as a mapping from IP to the powerset of $IR \times IR$. Given $p \in IP$, $IEXT(I(p))$ is the set of pairs $\langle x, y \rangle | x, y \in IR$ for which the relation p holds, that is, for which $\langle x, p, y \rangle$ is a statement in the RDF graph.

For an *RDF interpretation*, the following semantic condition holds[4]:

- $x \in IP$ if and only if $\langle x, I(\texttt{rdf:Property}) \rangle \in IEXT(I(\texttt{rdf:type}))$

Additionally, we define υ as a 'null' value, that is $I(x) = \upsilon$ if no value is assigned to x in the current interpretation. We will first characterize SeRQL in terms of RDF only, i.e. give an RDF interpretation. See table 4.1.

An extended interpretation takes into account RDF Schema semantics. For an *RDFS interpretation* the following semantic conditions hold in addition to those specified by an RDF interpretation (cf. [Hayes, 2004]):

- $x \in ICEXT(y)$ if and only if $\langle x, y \rangle \in IEXT(I(\texttt{rdf:type}))$
- $IC = ICEXT(I(\texttt{rdfs:Class}))$
- $IR = ICEXT(I(\texttt{rdfs:Resource}))$
- $LV = ICEXT(I(\texttt{rdfs:Literal}))$
- if $\langle x, y \rangle \in IEXT(I(\texttt{rdfs:domain}))$ and $\langle u, v \rangle \in IEXT(x)$ then $u \in ICEXT(y)$

[4] Other conditions also hold, see [Hayes, 2004], but these are not relevant for this discussion

`{x} p {y}`	$\{\langle x,p,y\rangle \mid \langle x,y\rangle \in IEXT(I(p))\}$
`{x} p {y};` `q {z}`	$\{\langle x,p,y\rangle \mid \langle x,y\rangle \in IEXT(I(p))\} \cup$ $\{\langle x',q,z\rangle \mid \langle x',z\rangle \in IEXT(I(q))\} \wedge x = x'$
`{x} p {y,z}`	$\{\langle x,p,y\rangle \mid \langle x,y\rangle \in IEXT(I(p))\} \cup$ $\{\langle x',p',z\rangle \mid \langle x',z\rangle \in IEXT(I(p'))\} \wedge x = x' \wedge p = p'$
`[{x} p {y}]`	$\{\langle x,p,y\rangle\}$ for which, depending on which variables are undefined, the following conditions hold: case 1: $I(x), I(p), I(y) \neq \upsilon$: $\langle x,y\rangle \in IEXT(I(p))$ case 2: $I(x) = \upsilon, I(p), I(y) \neq \upsilon$: $\nexists x' \mid \langle x',y\rangle \in IEXT(I(p))$ case 3: $I(x), I(p) = \upsilon, I(y) \neq \upsilon$: $\exists p' \mid \langle x',y\rangle \in IEXT(I(p'))$ case 4: $I(x), I(y) = \upsilon, I(p) \neq \upsilon$: $IEXT(I(p)) = \emptyset$ case 5: $I(p) = \upsilon, I(x), I(y) \neq \upsilon$: $\nexists p' \mid \langle x,y\rangle \in IEXT(I(p'))$ case 6: $I(p), I(y) = \upsilon, I(x) \neq \upsilon$: $\nexists p' \mid \langle x,y'\rangle \in IEXT(I(p'))$ case 7: $I(y) = \upsilon, I(x), I(p) \neq \upsilon$: $\nexists y' \mid \langle x,y'\rangle \in IEXT(I(p))$

Table 4.1: RDF interpretation of basic path expressions

- if $\langle x,y\rangle \in IEXT(I(\texttt{rdfs:range}))$ and $\langle u,v\rangle \in IEXT(x)$ then $v \in ICEXT(y)$
- $IEXT(I(\texttt{rdfs:subPropertyOf}))$ is transitive and reflexive on IP
- if $\langle x,y\rangle \in IEXT(I(\texttt{rdfs:subPropertyOf}))$ then $x,y \in IP$ and $IEXT(x) \subset IEXT(y)$
- if $x \in IC$ then $\langle x,IR\rangle \in IEXT(I\texttt{rdfs:subClassOf})$
- $IEXT(I(\texttt{rdfs:subClassOf}))$ is transitive and reflexive on IC
- if $\langle x,y\rangle \in IEXT(I(\texttt{rdfs:subClassOf}))$ then $x,y \in IC$ and $IEXT(x) \subset IEXT(y)$
- if $x \in ICEXT(I(\texttt{rdfs:ContainerMembershipProperty}))$ then $\langle x, I(\texttt{rdfs:member})\rangle \in IEXT(I(\texttt{rdfs:subPropertyOf}))$
- if $x \in ICEXT(I(\texttt{rdfs:Datatype}))$ and $y \in ICEXT(x)$ then $\langle y, I(\texttt{rdfs:Literal})\rangle \in IEXT(I(\texttt{rdf:type}))$

In table 4.2, the extensions of the interpretations of SeRQL path expressions and functions that the RDFS semantics add are shown.

At first glance, the added interpretations for properties such as `rdf:type` may seem redundant, in light of the fact that the case is already covered by the general path expression `{x} p {y}`. However, these mappings are added to make it explicit that these properties use an *RDFS* interpretation, that is, the semantic conditions regarding a.o. reflexivity and transitivity of these particular properties are observed.

`{x} rdf:type {y}`	$\{\langle x,y\rangle \mid x \in ICEXT(y)\}$
`{x} serql:directType {y}`	$\{\langle x,y\rangle \mid x \in ICEXT(y) \wedge$ $(\nexists z \mid z \neq y \wedge$ $x \in ICEXT(z) \wedge$ $\langle z,y\rangle \in IEXT(I(\texttt{rdfs:subClassOf})))\}$
`{x} rdfs:subClassOf {y}`	$\{\langle x,y\rangle \mid \langle x,y\rangle \in IEXT(I(\texttt{rdfs:subClassOf}))\}$
`{x} serql:directSubClassOf {y}`	$\{\langle x,y\rangle \mid x \neq y \wedge$ $\langle x,y\rangle \in IEXT(I(\texttt{rdfs:subClassOf})) \wedge$ $(\nexists z \mid x \neq z \neq y \wedge$ $\langle x,z\rangle, \langle z,y\rangle \in IEXT(I(\texttt{rdfs:subClassOf})))\}$
`{p} rdfs:subPropertyOf {q}`	$\{\langle p,q\rangle \mid \langle p,q\rangle \in IEXT(I(\texttt{rdfs:subPropertyOf}))\}$
`{p} serql:directSubPropertyOf {q}`	$\{\langle p,q\rangle \mid p \neq q \wedge$ $\langle p,q\rangle \in IEXT(I(\texttt{rdfs:subPropertyOf})) \wedge$ $(\nexists r \mid p \neq r \neq q \wedge$ $\langle p,r\rangle, \langle r,q\rangle \in IEXT(I(\texttt{rdfs:subPropertyOf})))\}$

Table 4.2: RDFS interpretation of basic path expressions

4.4.2 Functions

Datatypes, operators and functions are strongly interdependent, and to interpret function behaviour in SeRQL formally, we need to summarize how RDF itself handles datatypes. The following is summarized from [Hayes, 2004].

RDF provides for the use of externally defined datatypes identified by a particular URI reference. In the interests of generality, RDF imposes minimal conditions on a datatype.

The semantics for datatypes as specified by the model theory is minimal. It makes no provision for associating a datatype with a property so that it applies to all values of the property, and does not provide any way of explicitly asserting that a blank node denotes a particular datatype value.

Formally, a datatype d is defined by three items:

1. a non-empty set of character strings called the lexical space of d;
2. a non-empty set called the value space of d;
3. a mapping from the lexical space of d to the value space of d, called the lexical-to-value mapping of d.

The lexical-to-value mapping of a datatype d is written as $L2V(d)$.

SeRQL supports a set of functions and operators. These functions and operators can be used as part of the boolean constraints in the `where`-clause. Since these functions and operators deal with literal values that can be typed, we use the notion of an *XSD-interpretation* of a vocabulary V as specified in the RDF Semantics. An XSD-interpretation of a vocabulary V is an RDFS-interpretation of V for which the following additional constraints hold (see [Hayes, 2004] for a detailed explanation):

- D contains the set of all pairs of the form $\langle$`http://www.w3.org/2001/XMLSchema#sss`$, sss\rangle$, where `sss` is a built-in datatype named `sss` in XML

Schema Part 2: Datatypes [Biron and Malhotra, 2001], and listed in [Hayes, 2004], section 5.1.

- if $\langle a, x\rangle \in D$ then $I(a) = x$.
- if $\langle a, x\rangle \in D$ then $ICEXT(x)$ is the value space of x and is a subset of LV.
- if $\langle a, x\rangle \in D$ then for any typed literal `"sss"^^ddd` in V with $I(ddd) = x$, if sss is in the lexical space of x then $IL(\text{"}sss\text{"}\hat{}\hat{}ddd) = L2V(x)(sss)$, otherwise $IL(\text{"}sss\text{"}\hat{}\hat{}ddd) \notin V$
- if $\langle a, x\rangle \in D$ then $I(a) \in ICEXT(I(rdfs : Datatype))$

We provide a mapping for SeRQL functions in table 4.3.

`isResource(r)`	`true` if $I(r) \in IR$; `false` otherwise
`isLiteral(l)`	`true` if $I(l) \in LV$; `false` otherwise
`label("sss")`	$\{sss \mid I(\text{"}\texttt{sss}\text{"}) \in LV\}$
`label("sss"@lll)`	$\{sss \mid I(\text{"}\texttt{sss}\text{"}\texttt{@lll}) \in LV\}$
`label("sss"^^ddd)`	$\{sss \mid I(\text{"}\texttt{sss}\text{"}\hat{}\hat{}\texttt{ddd}) \in LV\}$
`datatype("sss"^^ddd)`	$\{ddd \mid I(\text{"}\texttt{sss}\text{"}\hat{}\hat{}\texttt{ddd}) \in LV\}$
`language("sss"@lll)`	$\{lll \mid I(\text{"}\texttt{sss}\text{"}\texttt{@lll}) \in LV\}$

Table 4.3: interpretation of SeRQL functions

4.4.3 Reducing Composed Expressions

In the previous sections we have seen how basic SeRQL expressions are formally interpreted. In this section, we show how composed path expressions can be reduced to semantically equivalent sets of basic path expressions and boolean constraints by means of a simple substitution.

Definition 2 (Path expression) *A* path expression *is of the form* $\langle n_0, e_0, n_1, e_1, n_2, ...e_{i-1}, n_i\rangle$, *where* i *is the length of the path expression, and where* $n_0..n_i$ *are nodes in the RDF graph and* $e_0..e_{i-1}$ *are directed edges. Each directed edge* e_k *has as source node* n_k *and as target node* n_{k+1}.

Definition 3 (Basic path expression) *A* basic path expression *is a path expression of length 1.*

As an example, the SeRQL construction `{x} p {y}` corresponds to the general form $\langle n_0, e_0, n_1\rangle$, and is a basic path expression.

A path expression of length $i > 1$ can be reduced to two path expressions, one of length $i - 1$ and one of length 1, as shown in table 4.4.

By recursively applying these substitutions to any path expression of length > 1 it is possible to reduce an arbitrary length composed path expression to a set of basic path expressions and boolean constraints. Thus, any complex SeRQL query can be normalized to a form consisting only of a set of basic path expressions and boolean constraints.

composed expression	substituted expressions	constraints
$\langle n_0, e_0, n_1, ..., n_{i-1}, e_{i-1}, n_i \rangle$	$\langle n_0, e_0, n_1, ..., n_{i-2}, e_{i-2}, n_{i-1} \rangle$, $\langle n'_{i-1}, e_{i-1}, n_i \rangle$	$n_{i-1} = n'_{i-1}$
$\langle n_0, e_0, n_1, e_1, n_2 \rangle$	$\langle n_0, e_0, n_1 \rangle$, $\langle n'_1, e_1, n_2 \rangle$	$n_1 = n'_1$

Table 4.4: Breaking up composed path expressions

Branching path expressions, multi-value node expressions and path expressions involving reification can also always be reduced to a set of basic expressions. We will prove this for branching path expressions, the proofs for the other two forms is analogous.

Theorem 1 *Any branching path expression p of the form* `{x} p {y}; q {z}` *can be reduced to a semantically equivalent set of basic path expressions.*

> **Proof:** By definition, the branching expression is syntactically equivalent to the two basic expressions `{x} p {y}, {x} q {z}` (see section 4.3.2). The first of these is defined as $\{\langle x, p, y\rangle | \langle x, y\rangle \in IEXT(I(p))\}$ (table 4.1). The second is defined as $\{\langle x, q, z\rangle | \langle x, z\rangle \in IEXT(I(q))\}$. The union of these two sets can be expressed as $\{\langle x, p, y\rangle | \langle x, y\rangle \in IEXT(I(p))\} \cup \{\langle x', q, z\rangle | \langle x', z\rangle \in IEXT(I(q))\} \wedge x = x'$, which is by definition (see table 4.1) equivalent to the definition of the branching path expression. □

4.5 SeRQL in Practice

SeRQL has recently become the default query language of the Sesame system. As such, it is being used by numerous developers and researchers in a wide variety of settings and domains. In this section, we will give a few brief examples of such use cases.

4.5.1 Querying Heterogeneous Data: FOAF

The Friend-Of-A-Friend (FOAF) project [5] is an inititiative to specify an RDF vocabulary for modeling a distributed social network. The idea is that every FOAF user can create and maintain an RDF file that uses this FOAF vocabulary to define something about this user: names, phone numbers, interests, and most importantly: which people you know. In this fashion, a huge distributed social network has come into existence, of people linked to each other through the `foaf:knows` relation.

Apart from a few core properties, however, the schema of FOAF is very loosely defined and constantly changes. The result of this is that some FOAF profiles contain much more information than others, and that sometimes different property names are used for what is essentially the same relation (for example, for the last name of people, `foaf:surname` `foaf:familyname` and `foaf:lastname` are in use, and some people do not bother with it at all and simply use `foaf:name` to record their full name only).

[5] See `http://www.foaf-project.org/`

When querying large amounts of aggregated FOAF data, the query language needs to be powerful enough to take such variations into account. As an example, a simple FOAF browser [6] that displays for a person (identified by the number '101' in this example) a person's name, e-mail address and picture, uses the following SeRQL query:

```
select FirstName, LastName, Name, Email, PictureLink
from   {x} rdf:type {foaf:Person};
          [foaf:name {Name}];
          [firstNameProp {FirstName}];
          [lastNameProp {LastName}];
          [foaf:email {Email}];
          [imageProp {PictureLink}];
where (firstNameProp = foaf:firstName or firstNameProp = foaf:givenName)
      and (lastNameProp = foaf:lastName or lastNameProp = foaf:surname)
      and (imageProp = foaf:depiction or imageProp = foaf:image)
      and x = 101
```

In this query, optional path expressions are used for almost every property of this person, to cope with the huge differences between different FOAF sources: not only may certain properties be undefined (for example, the e-mail address), but also several syntactical varieties of the same property need to be taken into account (for example, for the first name).

4.5.2 Using Tranformation Queries as Rules

Relational composition is a feature of ontology modeling that is not captured in the current specifications of RDF Schema, or even of the OWL ontology language. However, often rule languages can be seen as complementary to the modeling language. In the case of SeRQL, the graph transformation-type query can be seen as a simple rule. It can be used for relational composition. For example, assume we wish to express that person A is an uncle of person B if A has a brother who is a parent of B. This can be captured in a SeRQL query as follows:

```
construct {A} fam:uncleOf {B}
from      {A} fam:brotherOf {} fam:parentOf {B}
```

More generally speaking, SeRQL transformation queries can be used to express general entailment rules. For example, the transitivity of the `rdfs:subClassOf` relation is expressed by the following SeRQL query:

```
construct {A} rdfs:subClassOf {B}
from      {A} rdfs:subClassOf {} rdfs:subClassOf {B}
```

Currently, a prototype version of a custom inferencer for the Sesame framework is under developement that can be fed a set of SeRQL queries to use as entailment rules.

4.5.3 Defininging Views: the SWAP Project

The Semantic Web and Peer to Peer (SWAP) project [Broekstra et al., 2003] is a European IST project that aims to combine technologies from the areas of Ontology and P2P. The

[6] See `http://semwweb.cs.vu.nl/foaf-browser`

SWAP system is a decentralized environment in which peer nodes communicate and share knowledge, using RDF as the basic language. Each peer node in the SWAP system has a local repository, in which both local knowledge and knowledge obtained from other peers is stored, in RDF. A user interface allows users to edit, browse and query this knowledge. The de-facto standard query language in the SWAP system is SeRQL.

A problem in the SWAP system is that management metadata (sources of information, confidence ratings, etc.) are present in the same repository as the actual data. To allow convenient user access to the knowledge, SWAP employs definable views on top of the repository. These views are defined using the management metadata, but they only contain the domain knowledge. SWAP defines views by using SeRQL construct-queries that retrieve and transform relevant subgraphs from the repository.

For example, the following SeRQL query is used to construct the view that describes the expertise of known peers. This knowledge is not explicitly represented in the repository, but for every piece of knowledge an associated peer is known:

```
construct
     {P} view:knowsAbout {C}
from
     {{C} rdf:type {rdfs:Class}} swap:hasSwabbi {} swap:hasPeer {} swap:hasLabel {P}
```

(The 'hasSwabbi' property associates a particular domain knowledge statement with an object known as a 'Swabbi'. This Swabbi object then is a placeholder for all the relevant management metadata for this particular statement. See [Broekstra et al., 2003] for details.)

In general, the use of transformation queries in the SWAP context is invaluable, as the information shared between peers consists mainly of RDF models. Since query answers are RDF models themselves, this allows easy integration of knowledge from other peers in a particular peer's own knowledge repository.

4.5.4 Mapping Vocabularies: the DOPE Project

The aim of the DOPE project (Drug Ontology Project for Elsevier) [Stuckenschmidt et al., 2004a] is to investigate the possibility of providing access to multiple information sources in the area of life sciences, through a single interface. The prototype system that was developed allows thesaurus-driven access to heterogeneous and distributed data, based on the RDF model.

In figure 4.1, the architecture of the DOPE system is shown. Central to the architecture is a mediator, that functions as a central access point for queries posed by the user interface and distributes the query over the distributed data sources. In the prototype, this mediator has been realized using Sesame's SAIL (Storage And Inference Layer) API, on top of which a SeRQL query engine func-

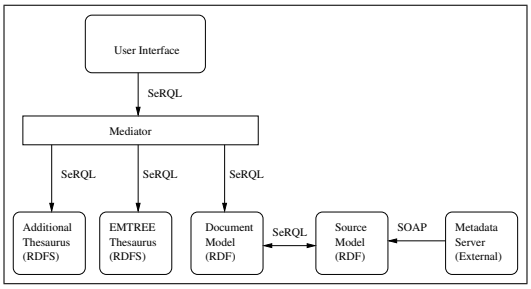


Figure 4.1: DOPE architecture

tions as the entry point for the user interface.

The *Metadata Server* is a repository of information that is not equipped with RDF i/o. However, it does have a SOAP interface. Therefore, an extractor component is deployed which, through use of the SOAP interface, converts the available information in an RDF format that is a 1:1 mapping to the model as it is represented internally in the Metadata server. This model is referred to as the *source model*. The data from this source is now in RDF, but not in the terminology that user queries are formulated in. Therefore, a transformation takes place from the physical model to a *document model*, using SeRQL construct-queries to define and perform the transformation.

4.6 Future Work

The SeRQL proposal as documented in this chapter has turned out to be a powerful and useful tool in querying RDF data. Nevertheless, practical experience as well as analysis of the language's capabilities (see also chapter 3) has enabled us to identify several shortcomings of the query language that could be captured in future extensions.

Currently, development work is being done on implementing several such extensions in Sesame's SeRQL engine. Specifically, the identified areas for extensions are:

- aggregation operations, such as `count()`, `max()`, `avg()`, `min()`.
- result grouping.
- algebraic operations, such as union, intersection and difference.
- query nesting and quantification.

Most if not all of these are straightforward extensions of the existing engine, and many of them have indeed been implemented since the time of writing. In the next few sections, we will look at several of these extensions in more detail.

4.6.1 Aggregation and Grouping

Aggregation and grouping are important operations in analyzing and sorting data, and are a integral part of SQL. Nevertheless, virtually none of the currently existing RDF query language proposals seem to offer support for it (see chapter 3).

The current SeRQL proposal will be extended with aggregation functions `count()`, `sum()`, `max()`, `min()` and `average()` as well as a grouping operations `group by`. This will, for example, enable queries such as:

```
select publication, count(author) as numberOfAuthors
from {publication} rdf:type {my:Publication};
                   my:author {author}
```

which retrieves the number of authors per publication.

4.6.2 Algebraic operations

Basic algebraic operations such as union, difference and intersection are essential building blocks for a query language in terms of relational completeness. In a prototype implementation SeRQL has been extended with such operations. For example, the use case *Difference* in chapter 3 was the query *"return all topic titles that are not also titles of pubications"*. This would be expressed in SeRQL as follows:

```
 select title
 from {} rdf:type {t:Topic};
         rdfs:label {title}
minus
 select title
 from {} rdf:type {t:Publication};
         rdfs:label {title}
using namespace t = ...
```

4.6.3 Query Nesting and Quantification

A set membership operator, `in`, will be introduced that can be used to check value occurrence in a nested subquery. Also, quantification operations such as `any` and `all` give a very powerful new way of comparing values, not just for a particular instantiation, but comparing over a set of instantatiations.

4.7 Conclusions

In the previous sections, we have given an overview of the SeRQL query language, and we have demonstrated how SeRQL fulfills a set of key requirements for RDF query languages. We have provided the basic syntax and a formal model.

SeRQL is an attempt to come to an RDF query language that satisfies necessary general requirements on such a language without adding unnecessary bloat. Specifically, SeRQL has been designed to be fully compatible with the RDF specifications, to be easy to read and write by humans while at the same being easy to process and produce in an automated fashion. Most of the features of SeRQL are not new, but we believe that SeRQL is the first proposal that combines all these requirements in a single language and the only such proposal that has been implemented succesfully and is being used succesfully.

Future work on the development of SeRQL as a language will focus on adding useful and necessary functions and operators demanded by the user community, as well as encouraging other developers to implement engines that support this language.

Part II

Implementing Middleware for the Semantic Web

Chapter 5

Sesame: an RDF Framework

In the previous chapters we have seen overviews of languages for modeling and manipulating ontologies. In this chapter, we will look at programmatic tooling for such languages. Specifically, we will introduce and discuss Sesame, which is a storage and querying framework for RDF and RDF Schema.

The work presented in this chapter is an adaption and extension of [Broekstra et al., 2002].

5.1 Introduction

As described in chapter 2, the Resource Description Framework (RDF) [Lassila and Swick, 1999, Beckett, 2004] is a W3C Recommendation for the formulation of metadata on the World Wide Web. RDF Schema [Brickley and Guha, 2000, Brickley and Guha, 2004] (RDFS) extends this standard with the means to specify domain vocabulary and object structures. These techniques will enable the enrichment of the Web with machine-processable semantics, thus giving rise to what has been dubbed the Semantic Web.

We have developed Sesame, a Java framework for storage and quering of RDF and RDFS information. Sesame is being developed by Aduna[1], originally as part of the European IST project On-To-Knowledge[2] [Fensel et al., 2000b], later as an independent open-source project under the name openRDF.org[3]. Sesame allows persistent storage of RDF data and schema information, and provides access methods to that information through export and querying modules. It features ways of caching information and offers support for concurrency control.

This chapter is organized as follows. In section 5.2, we introduce the general Sesame architecture and discuss its storage and access APIs. In section 5.3, Sesame's generic query model is discussed. In section 5.4, we discuss various storage backends in more

[1] See `http://aduna.biz/`

[2] On-To-Knowledge (IST-1999-10132). See `http://www.ontoknowledge.org/`

[3] See `http://www.openrdf.org`

detail. In section 5.6 we discuss future improvements to the framework, and finally we present our conclusions in section 5.8.

5.2 The Sesame Architecture

Sesame is designed as a layered architecture where each level abstracts away further from physical storage and adds more conceptual notions regarding RDF.

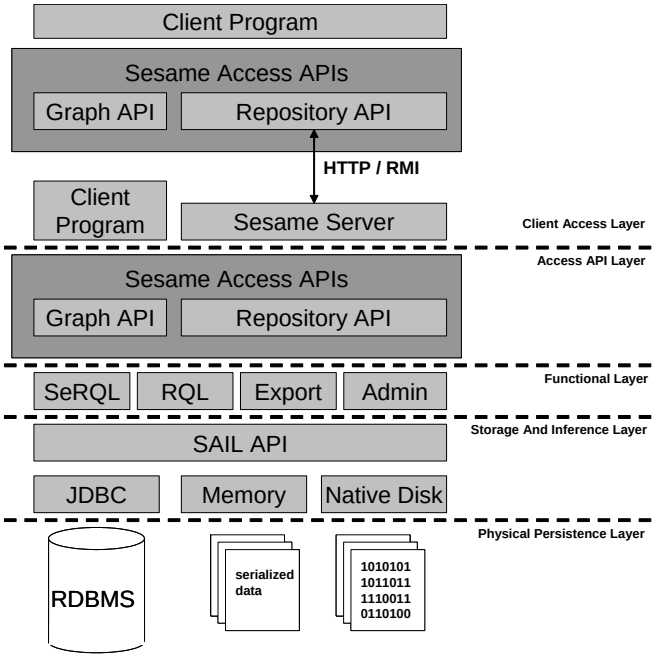


Figure 5.1: Sesame Architecture

Figure 5.1 gives a schematic overview of the Sesame architecture. At the bottom, we find the physical storage of the data, which can be an in-memory model, storage on disk in a binary format, or an RDBMS. The *Storage And Inference Layer* (SAIL) is an API that abstracts away from the details of physical storage and provides generalized storage and retrieval methods, as well as inferencing support, to the higher layers.

In the next layer we find the *functional modules*. These provide higher-level functions based on the storage and retrieval primitives in the underlying SAIL, such as adding RDF files, querying using the SeRQL (see chapter 4), RQL or RDQL query languages, and so forth.

On top of that, Sesame's *Access APIs* provide a set of interfaces for transparent access to Sesame's functionality. The API is designed in such a way that switching from local

to remote access demands minimal effort on the part of the client. Remote access is supported through either HTTP or RMI.

Sesame can be deployed as a normal Java library, embedded in a client application, but it can also be used as a standalone server. The *Sesame Server* is a Java Servlet application that can be deployed in a Servlet Container such as Apache Tomcat[4], and which enables one to connect to this server and store and query RDF over the Web.

In the next sections, we will look in more detail at several of Sesame's components.

5.2.1 The SAIL API

The Storage And Inference Layer, or SAIL, is a set of interfaces that abstracts away from the details of storage and allows the functional modules that operate on it to be agnostic to the chosen storage model. The main design principles of the SAIL API are that the API should:

- define a basic interface for storing RDF and RDFS in, and retrieving and deleting RDF and RDFS from (persistent) repositories.
- abstract from the actual storage mechanism; it should be applicable to RDBMSs, file systems, or in-memory storage, for example.
- be usable on low end hardware like PDAs, but also offer enough freedom for optimizations to handle huge amounts of data efficiently on e.g. enterprise level database clusters.
- be extendable to other RDF-based languages like DAML+OIL [Horrocks et al., 2001] or OWL [Dean and Schreijber, 2004].

Other proposals for RDF APIs are currently under development. The most prominent of these are the Jena toolkit [Carrol and McBride, 2001] and the Redland Application Framework [Beckett, 2001]. SAIL shares many characteristics with both approaches.

An important difference between these two proposals and SAIL, is that the SAIL API specifically deals with RDFS on the retrieval side: it offers methods for querying class and property subsumption, and domain and range restrictions. In contrast, both Jena and Redland focus exclusively on the RDF triple set, leaving interpretation of these triples as an excercise to the user. In SAIL, these RDFS inferencing tasks are handled internally. The main reason for this is that there is a strong relationship between the efficiency of the inferencing and the actual storage model being used. Since any particulary SAIL implementation has a complete understanding of the storage model (e.g. the database schema in the case of an RDBMS), this knowledge can be exploited to infer, for example, class subsumption more efficiently.

Another difference between SAIL and other RDF APIs is that SAIL is considerably more lightweight: only four basic interfaces are pre-defined, offering basic storage and retrieval functionality and transaction support, but not much beyond that. We feel that in

[4] See `http://jakarta.apache.org/tomcat`

some applications such minimality may be preferable to an API that has more features, but is also more complex to understand and implement.

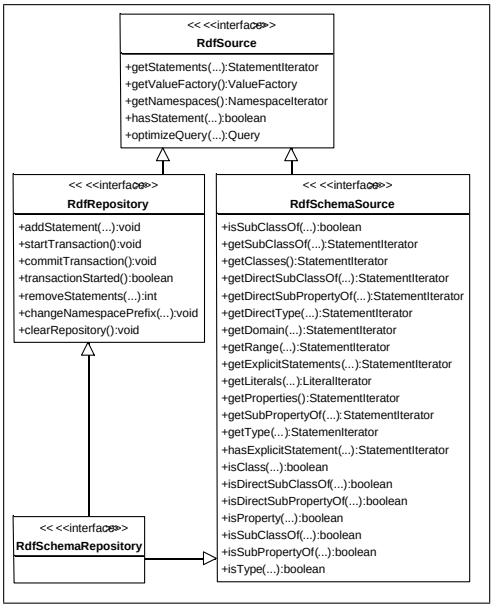


Figure 5.2: The SAIL interface hierarchy

The SAIL API has been designed to allow maximum flexibility in the storage mediums chosen. To this end, the API consists of four interfaces that divide operations across two orthogonal dimensions: read- vs. write-operations, and RDF- vs. RDF Schema operations (see figure 5.2).
The topmost interface, `RdfSource`, provides basic RDF read operations, such as, for example `getStatements(subject, predicate, object)`, which returns (lazy) iterators of statements that match the provided (subject, predicate, object) pattern. Two subsequent interfaces specialize, one (`RdfRepository`) in the direction of providing write access (such as adding and removing statements), and the other (`RdfSchemaSource`) in the direction of providing RDF Schema operations, such as retrieving subClasses and -properties. The final interface, `RdfSchemaRepository` provides no additional methods but unifies the two orthogonal specializations in a single interface.

The chosen separation of functionality makes it possible to deploy the Sesame framework on top of a wide variety of systems. For full storage, retrieval and inferencing support, the `RdfSchemaRepository` can be implemented. Sesame supports implementations of this interface that work on top of relational databases, or in main memory. However, the flexibility of the APIs allows connecting arbitrary sources to the system. An existing legacy database or any other source of information can be wrapped in a SAIL implementation, and depending on the situation, a choice can be made whether Sesame is allowed to modify the source or not. We will describe several examples of this in later chapters.

As can be seen, the SAIL architecture that was developed fullfills the design set for the API that we set out earlier:

- in a layered set up full functionality for storage and retrieval of RDF, through very basic operations, is available.
- the separation of functionality across two orthogonal dimensions, as described, makes the architecture ideally suited to be deployed on any storage backend, thus making the API an ideal abstraction layer for storage.
- the minimal nature of the API (in terms of methods), combined with the consistent

use of streaming data passing to minimize memory footprint, makes the SAIL suited for low end hardware.

- the fact that the SAIL consists of several interfaces makes future semantic extension possible: OWL-specific functionality could be defined in a separate interface that would take its place in the SAIL interface hierarchy. This will not affect existing implementations.

Stacking SAILs

An important feature of the SAIL is that it is possible to put one on top of the other (see figure 5.3). The SAIL at the top can perform some action when the modules make calls to it, and then forward these calls to the SAIL beneath it. This process continues until one of the SAILs finally handles the actual retrieval request, propagating the result back up again.
We implemented a SAIL that caches all schema data in a dedicated data structure in main memory. This schema data is often very limited in size and is requested very frequently. At the same time, the schema data is the most difficult to query from a DBMS because of the transitivity of the subClassOf and subPropertyOf properties. This schema-caching SAIL can be placed on top of arbitrary other SAILs, handling all calls concerning schema data. The rest of the calls are forwarded to the underlying SAIL.
Another important task that can be handled by a SAIL is concurrency handling. Since any given query can be broken down into several operations on the SAIL level, it is important to preserve repository consistency over multiple operations. We implemented a SAIL that selectively blocks and releases read and write access to repositories, on a first come first serve basis. This setup allows us to support concurrency control for any type of repository.

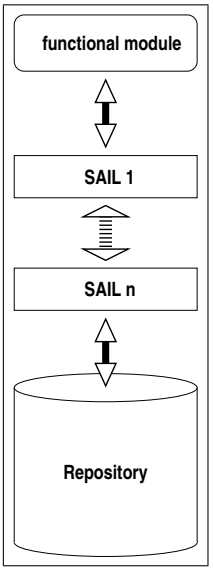


Figure 5.3: A Stack of SAILs

5.2.2 Functional Modules

The SAIL API presented in the previous section provides very basic methods for storing and retrieving RDF statements. To provide more high level functionality, several functional modules are deployed on top of the SAIL.

The current version of Sesame (1.1) contains the following functional modules:

- Query engines for the *SeRQL*, *RDQL* and *RQL* RDF query languages. The query engines translate an incoming query to a generic query object model (see section 5.3), evaluate the query and push the query result back to the query client through a `Listener` object.

- The *Admin module* offers functionality for adding RDF files to and removing data from a Sesame repository.

- The *Export module* offers functionality for retrieving the contents of a repository as a serialized file. Supported serializations include RDF/XML, N-Triples and Turtle[5]. Export can be limited to ontology or data only, and can include or exclude inferred statements.

5.2.3 The Access APIs

Sesame's access APIs can be divided in two parts: the *Repository API* and the *Graph API*. The former provides central high level access functions to Sesame repositories, the latter more fine-grained manipulation for RDF models. In practice, the two APIs complement each other and are often used together.

In figure 5.4, we see an overview of Sesame's Repository API.

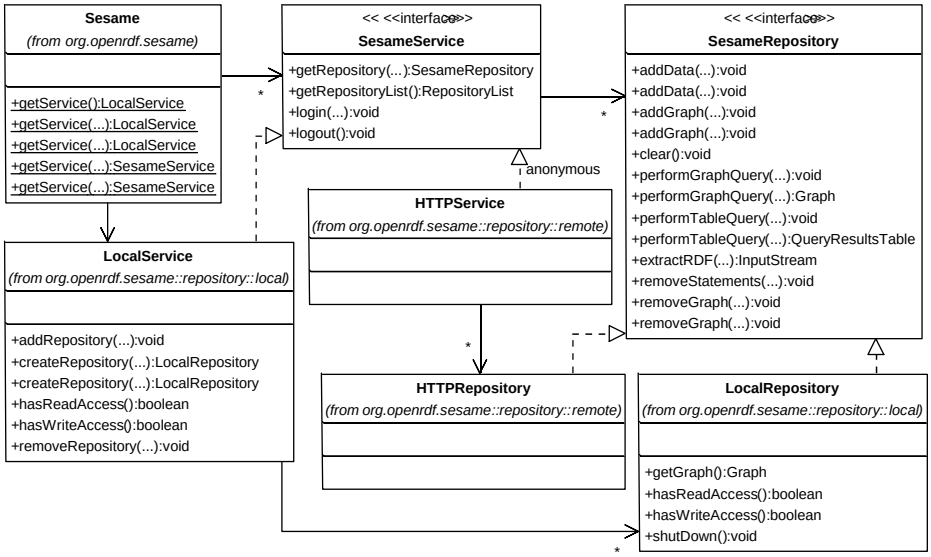


Figure 5.4: The Repository API

The central idea behind the Repository API is to provide high level access to Sesame repositories in a way that hides irrelevant detail from the client. The API has been set up to function identically for both remote access and local access. The `SesameRepository` interface defines operations for adding data, querying the repository, and removing data. These operations are high-level in the sense that they operate on the repository as a whole, and hide transaction management and other lower-level details from the client.

The Graph API (figure 5.5) complements the functionality of the Repository API by providing an object model for RDF graphs, statements, and statement components (i.e. URIs, blank nodes, and literals). Graphs are aggregations of statements which can be

[5]See `http://www.ilrt.bris.ac.uk/discovery/2004/01/turtle/`

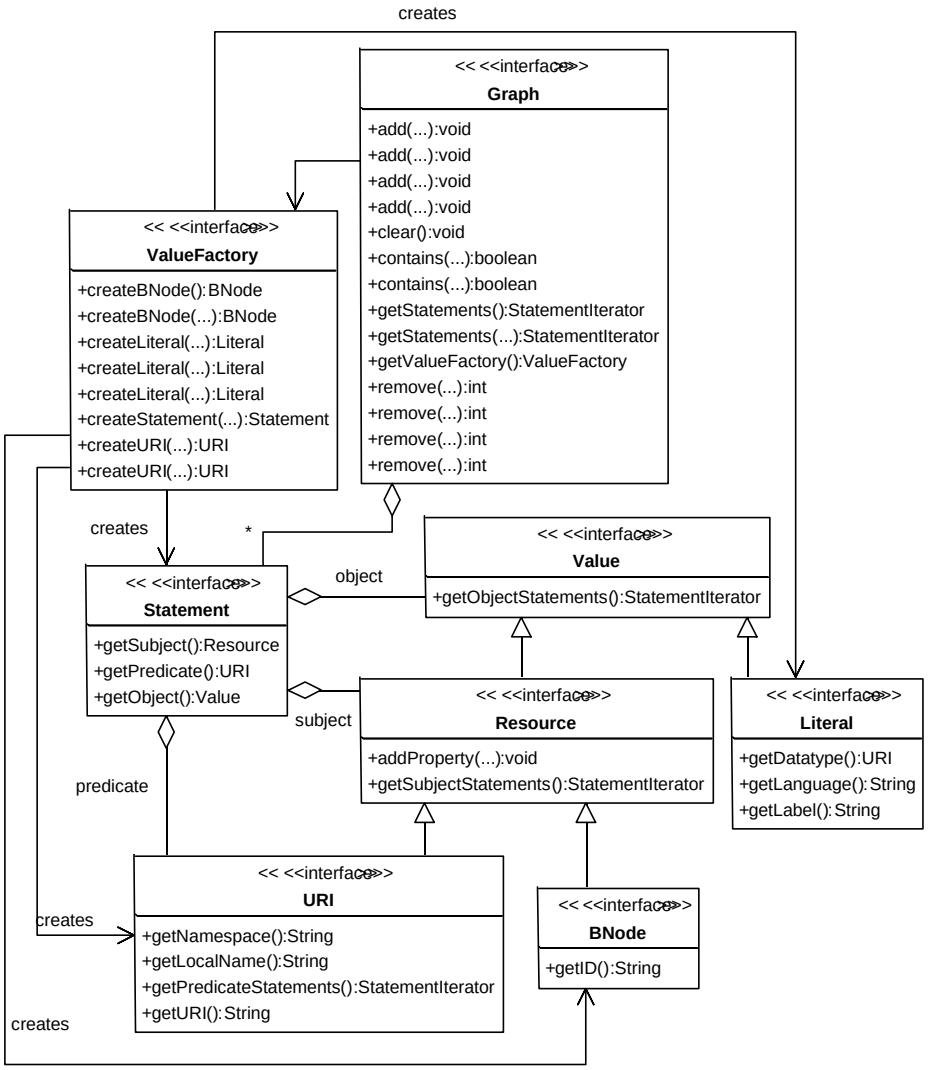


Figure 5.5: The Graph API

manipulated through add and remove operations, working on either sets of statements or individual statements. The graph API allows the client to do fine-tuned manipulation of the RDF stored in a repository, or even creating new RDF models on the fly without using a repository.

5.3 Querying in Sesame

The Sesame framework provides three query engines, one for each query language it implements. The query engine in actuality only parses each incoming query and builds a language-independent query object model out of it. This query object model can then evaluate itself by finding variable bindings through communication with the underlying SAIL.

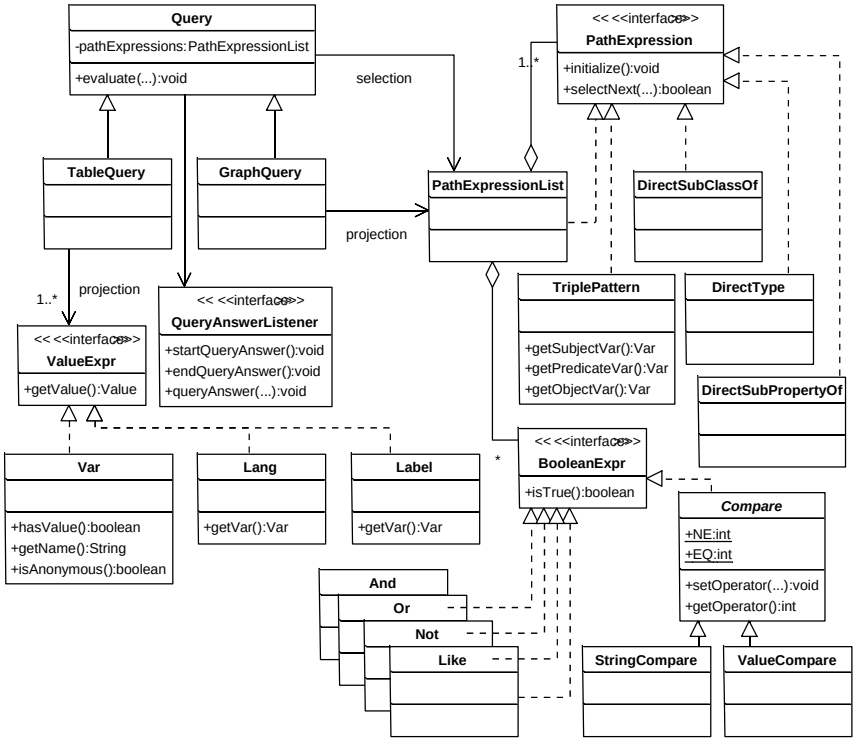


Figure 5.6: Sesame's Query Object Model

In figure 5.6, we see an overview of the Sesame query object model. The `Query` class is the generalized representation of a query. It has two specializations: `TableQuery` and `GraphQuery`. These specializations deal with the different types of query answer projection that Sesame supports: a `TableQuery` projects the query result into sets of variable bindings, whereas a `GraphQuery` retrieves graphs (i.e. sets of triples).

The query model is self-evaluating: when `Query.evaluate()` is invoked, this is propagated into the list of path expressions (through the `selectNext` method). Each `PathExpression` in the list tries to instantiate its connected variables to a new value. If this succeeds for the entire `PathExpressionList`, a new query result is found and reported back to the `QueryAnswerListener`.

Boolean expressions and comparison operators are part of the `PathExpressionList` as well. This approach allows the query optimizer to simply rearrange the order of the list to optimize query performance: a comparison operator is moved as near to the beginning of the list as possible (since it is an $O(1)$ operation, and prunes large portions of the evaluation space, it is both cheap and rewarding to evaluate such operations as early as possible), directly after the path expression(s) that initialize its operands.

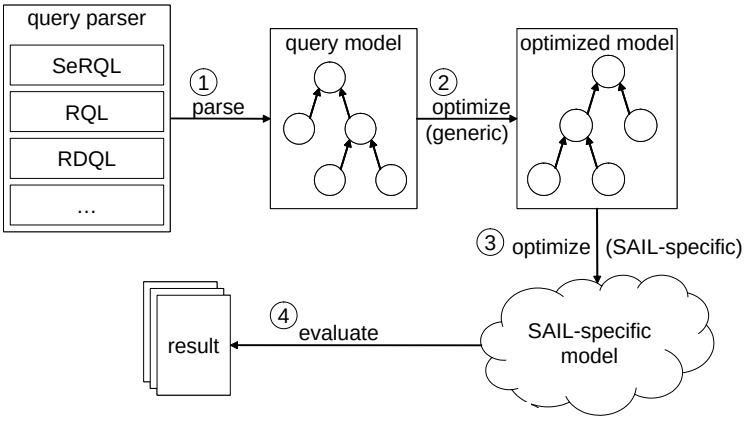


Figure 5.7: The four steps of query evaluation

In figure 5.7, it is illustrated how a query is evaluated in the framework. In step 1, the query parser takes a query in any language and translates it to a generic query model. In step 2, generic optimizations, such as path expression and boolean comparison rearrangement, are made. The result of this is a optimized model that uses a depth-first search strategy for evaluation. The third step consists of SAIL-specific optimizations: since the storage backend has more intimate knowledge of indexing structures and the way in which the data can be retrieved, it is allowed to further optimize the model. The result of this is completely open: if the SAIL chooses to ignore the optimizaton request, the original model is left intact. However, the SAIL can also choose to completely rearrange or even replace the object model, with a storage-specific query model. An example of this is the RDBMS SAIL, which completely discards the generic query object model and replaces it with a query representation that maps it directly to an SQL query. All that is required is that the object is an implementation of the generic `Query` interface from figure 5.6.

In step 4, the query object is allowed to evaluate itself. In the case of the generic model, this means that a depth-first search is initiated through the path expression lists: starting at the root (the first path expression of the query's from-clause), each expression evaluates itself and tries to instatiate its variables by calling appropriate information

retrieval methods on the underlying SAIL. The algorithm descends until all variables are instantiated. If an expression at some point in the descent can not find a legal instantiation, the algorithm backtracks to an earlier expression that shares a variable with the failed expression, and tries to find an alternative value. If at some point no further backtracking is possible, the query fails: no further query results can be found.

The query evaluation process of Sesame has the distinct advantage of being extremely flexible: practically any RDF query language can be mapped to the generic model, and evaluation of a query is guaranteed to work since the generic model has its own evaluation strategy. On the other hand, the model is flexible enough to allow specific stores to replace the generic model with a store-specific representation of the query (such as a direct mapping to SQL in the case of an RDBMS backend), for performance reasons.

5.4 Storage Backends

The Sesame framework supports several types of storage, each with their own unique characteristics. As explained in previous sections, the SAIL API abstracts away from these characteristics to allow each type of store to be used with any of Sesame's functional modules. In this section, we will take a closer look at the details of several of these storage backends.

5.4.1 The RDBMS Backend

Sesame's RDBMS backend uses a vertical storage model: each RDF statement is stored as a single row in a table. However, several extensions and optimizations have been made.

In figure 5.8 we see a representation of Sesame's RDBMS schema. The central table is Triples, which encodes each RDF statement as a row in the table, with seperate columns for the subject, predicate, and object. The additional columns are an internal unique statement identifier and a boolean marker that indicates if the statement was explicitly inserted or was inferred by the inferencer.

In order to minimize space requirements, Sesame uses mapping tables that map resource names (URIs) or literal strings to an integer number. This integer number is then used in all subsequent tables, saving duplication of potentially long character strings. The RDBMS SAIL caches the identifiers of resources and literals, thus avoiding potentially costly lookups in the mapping table(s).

Further optimizations are the introduction of several auxiliary tables, such as Class, Subclass, Subproperty, etc. In this case, we trade storage space for querying performance: the auxiliary tables duplicate information that is already present in the Triples table, but they make it possible for specialized queries, that involve RDF Schema semantics, to be evaluated much more quickly.

Apart from the table structure itself, several indexes are defined to speed up lookups. Here again a tradeoff has to be made: while indexes speed up lookups, they slow down insertion and require additional storage space.

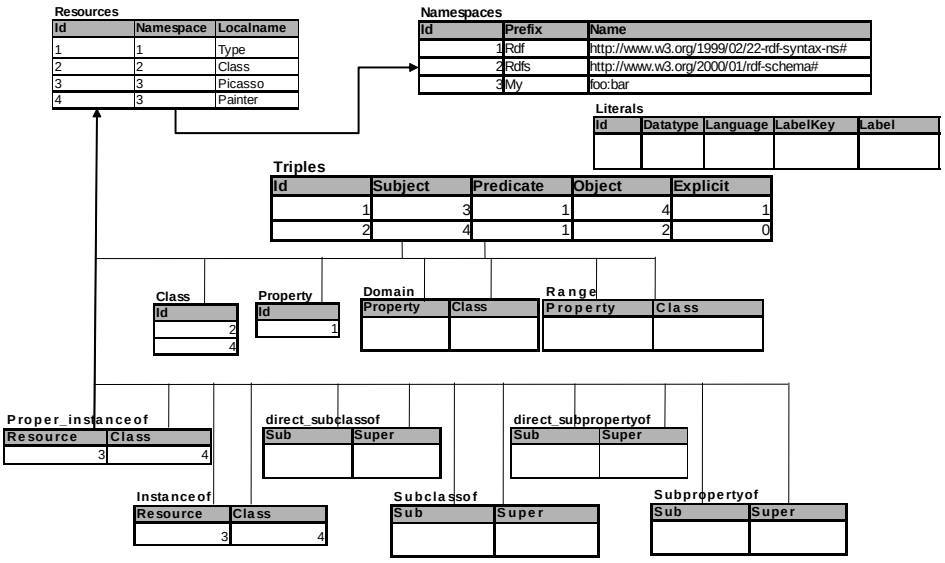


Figure 5.8: Sesame RDBMS Schema

5.4.2 The Object-Relational Backend

In earlier verions of the Sesame framework, an alternative storage model for databases, using Object-Relational features for relating data structures, was used.

In figure 5.9 we see a simplified[6] representation of the Object Relational Schema used by this backend. As can be seen, the schema is adaptive to the actual data being stored: each class and property is represented by its own table. The schema makes use of the Object-Relational notion of *subtables* to model the class and property subsumption hierarchy. Thus, the fact that `FamousWriter` is a subclass of `Writer` is reflected in the way the tables are related.

An advantage of this model is that the need for class-instance relation entailment is removed: the object-relational database takes care of all such inheritance entailments automatically. For example, an SQL query that retrieved all records in the `Writer` table would automatically also receive all records in the `FamousWriter` table.

However, a large disadvantage of the model is the fact that the schema is volatile: each time a new class or property is inserted, a new table has to be introduced. Also, the schema potentially creates an enormous amount of tables, each of which contains relatively few records. This hampers querying and indexing efficiency.

Another disadvantage is that the subtable relation does not completely match the semantics of the `rdfs:subClassOf` relation: object-relational subtables can only have

[6] In reality, the ORDBMS schema uses mappings from unique integer IDs to actual URIs, like in the relational schema. These tables are not present in the figure.

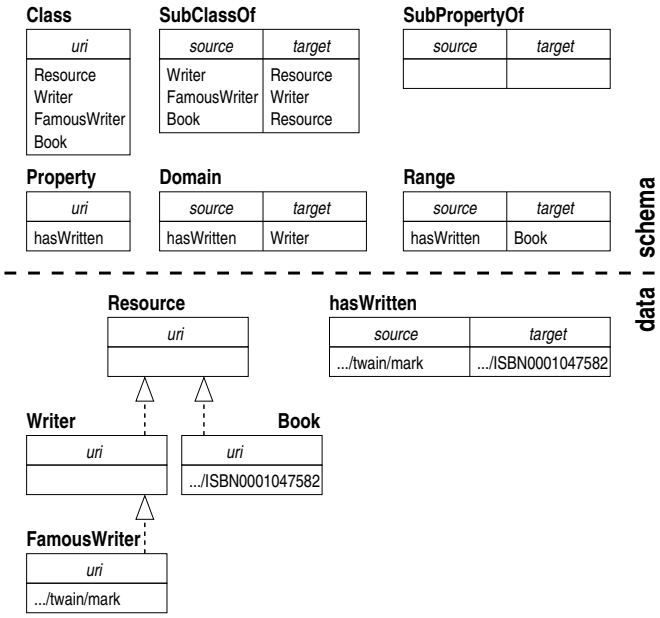


Figure 5.9: Sesame ORDBMS Schema (simplified)

a single parent, whereas RDF Schema allows any class to have multiple parents.

The observed disadvantages have led to the abandonment of the object-relational schema in newer versions of the Sesame framework.

5.4.3 Main Memory

As an alternative to a relational database, Sesame also supports storage in main memory. The advantages of this approach (as opposed to the relational database) are several:

- Installation and deployment of the Sesame framework no longer requires installation and configuration of a separate RDBMS.
- Insertion, inferencing and querying speeds are all improved significantly.

Of course, a big disadvantage of the approach is the cost of main memory storage. The chosen object model therefore tries to minimize the amount of memory needed.

The main memory store uses a bipartite graph model (see figure 5.10) to store RDF graphs. In this model, each resource (or literal) is represented by a unique object vertex, which is connected to every statement vertex in which it plays a role (thus, the bipartition in this case is the distinction between resource/literal vertices and statement vertices). The edge label identifies the role the resource/literal plays in the particular statement, i.e. subject, predicate, or object. For example, in figure 5.10, `statement01` represents the RDF statement `(foo:person001 rdf:type foo:Person)`.

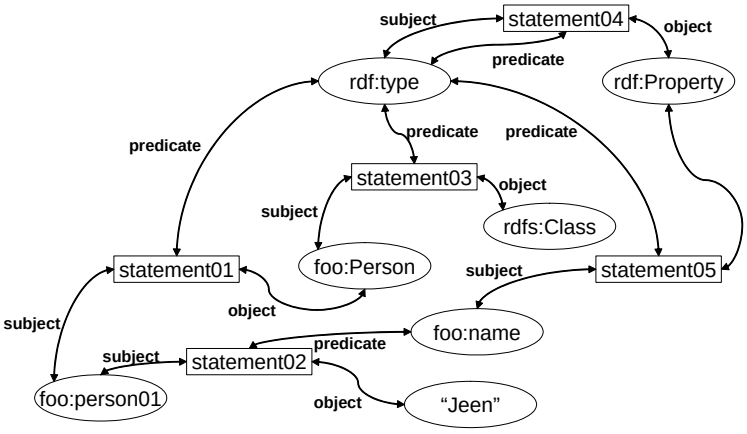


Figure 5.10: Bipartite graph representation of an RDF model

The advantage of using this representation is that it eliminates the need for object duplication: each resource has a single vertex representing it. In a 'normal' RDF graph, duplication of resources occurs whenever a resource plays the role of both a subject and a predicate (for example, the `rdf:type` relation in figure 5.10). Not only does this save space, it also allows faster retrieval as no duplication has to be taken into account during lookups. Additionaly, in [Hayes and Gutierrez, 2004] it is argued that bipartite graph representations have the advantage of mapping directly to standard graph-based algorithmic solutions to problems of querying and storing.

5.4.4 Native Disk Storage

Sesame's native disk storage SAIL is a relatively new storage backend, implemented with high scalability and performance, and low deployment overhead (e.g. no installation of third party software) as main design goals.

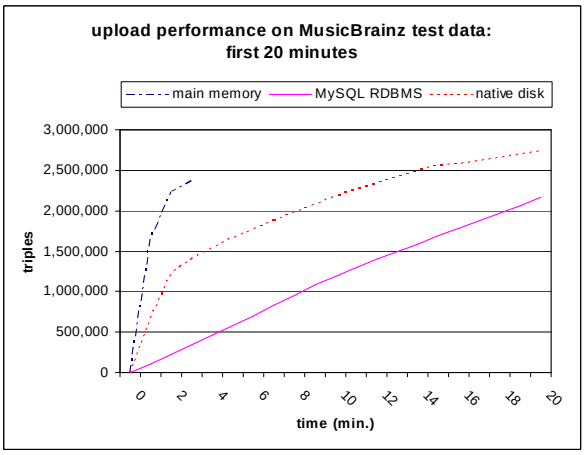
The native disk storage SAIL uses files for storing and querying the RDF data, using Java's new I/O classes ("nio", package java.nio) for accessing them. Nio offers low-level, high-performance file access and allows one to memory-map files relying on the native file system's page handling algorithm for (selective) caching and the like.

Common database techniques like B-trees, hash tables, etc., which are also stored on disk, are used to be able to quickly search through the stored RDF graph. The SAIL employs selective caching of (part of) the data in memory for increased retrieval performance.

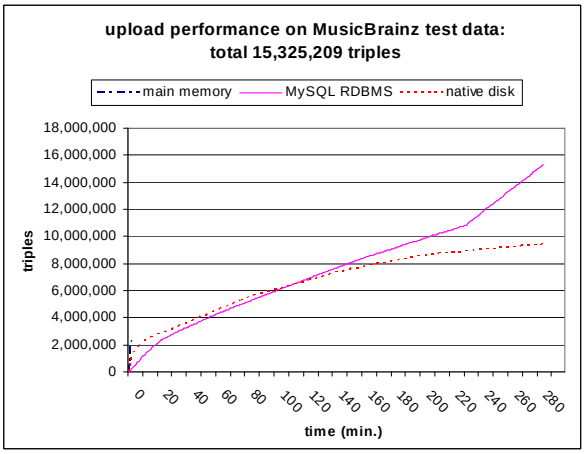
5.4.5 Performance

The availability of several alternative storage mechanisms begs the question which backend to choose in which use case. In the previous section, we have seen short descriptions

of each storage backend, including its design goals and main characteristics. To test whether these design goals are met by the implementation, several benchmark tests were performed.



(a) first 20 minutes



(b) complete upload

Figure 5.11: Upload performance of Sesame storage backends on MusicBrainz data

In figure 5.11, we see the upload performance of several storage backends.[7] Inferencing is not included here; for figures on inferencing performance see chapter 6. These figures were obtained over a single upload session of the MusicBrainz RDF dump, con-

[7]The Object-Relational schema is not included in these figures because it has been deprecated from the Sesame framework.

sisting of approximately 15 million triples. The measurements were obtained on a 2GHz Pentium 4 machine with 600MB RAM memory reserved for the Java heap.

In the first graph, we see that the initial performance of the main memory store is very high: it reaches 2 million triples in less than two minutes. However, in our test setup, the system ran out of sufficient memory shortly after reaching 2.4 million triples. Obviously, while much faster than the other stores, the scalability of the main memory store is limited to the amount of RAM memory available.

The native disk store performs significantly better than the RDBMS store for the first 1.5 million triples, but then performance degrades to about equal to that of the RDBMS store. In fact, in the second graph we can observe that while the upload speed of the RDBMS store is roughly linear (with a small performance decrease when reaching about 3 million), the upload speed of the native disk store steadily decreases. At the point where the RDBMS has completed the upload, the native store has completed only about 66%.

5.4.6 Discussion

The performance figures shown in the previous section clearly demonstrate that the main memory store is especially useful in cases where the data set is relatively small, in the order of 2 million triples. The native store is more suitable for larger data sets, up until about 6 million triples. If the data set becomes larger than 6 million, the RDBMS store, even though its initial upload performance is lower than the other two stores, eventually outperforms the native and main memory store, in terms of consistency of upload speed and scalability.

5.5 Inferencing Support

The Sesame framework has full support for the semantics of RDF and RDFS as described in [Hayes, 2004]. In terms of the architecture, support for these semantics is located in the SAIL layer, ostensibly in the `RdfSchemaSource` interface of the API (see figure 5.2).

The RDBMS and main memory store (the native store currently does not support inferencing) support inferencing through a forward chaining inferencer that does a pruned iterative sweep over the store, computes the closure and stores it in the repository. Thus, at query time, every inferencing task is reduced to a simple database lookup.

A second type of inferencer, the *custom inferencer*, is supplied as an alternative for default RDF Semantics reasoning. It can be used in combination with the RDBMS store and allows the user to specify his own entailment rules in an XML-based rule file. It uses a similar strategy for reasoning though: the closure is computed and stored.

In chapter 6, we will look at inferencing strategies in more detail.

5.6 Future Work

5.6.1 Transaction rollback support

While the SAIL API has support for transactions, it currently has no transaction rollback feature. Transaction rollbacks, especially in the case of uploading information, are crucial if we wish to guarantee database consistency. In the case of RDF uploads, transaction rollbacks can be supported at two levels:

- a single upload of a set of RDF statements can be seen as a single transaction, or alternatively, a single upload can be ”chunked” into smaller sets to support partial rollback when an error occurs during the upload session.
- a single RDFS statement assertion can been seen as a transaction in which several tables in the database need to be updated. From the user point of view, the schema assertion is atomic (”A is a class”), but from the repository point of view, it may consist of several table updates.

Both levels of transaction rollback support may help ensure database consistency. Together with the concurrency support already present in the Sesame system, this will help move Sesame towards becoming an ACID[8] compliant storage system (note that this can only be guaranteed if the platform used for storage supports it).

5.6.2 Generic Inferencers

In the current implementation, the inferencers are specific to a particular SAIL implementation: the RDBMS and main memory store each have their own inferencer.

However, the notion of the stacked SAIL lends itself particularly well to creating a more generic inferencer. The inferencer can in fact act as a filter between incoming calls from the functional modules and the eventual storage. This way, the inferencing strategy can be completely hidden: forward chaining inferencers will act on any commit operation by computing and storing the deductive closure, while a backward chaining inferencer can react directly to any querying operation.

5.6.3 OWL Reasoning

The current version of Sesame supports the RDF Schema semantics, and has an option for defining custom entailment rules. However, Sesame currently has no explicit support for the more expressive OWL ontology language [Dean and Schreijber, 2004].

Several reasoner tools that specifically deal with DL-style languages like OWL are available, such as Racer[9]. A recent development is the DIG interface [Bechhofer, 2003], a standardized XML interface to Description Logics systems. Through the DIG interface client tools may communicate with DL reasoners.

[8] *Atomicity, Concurrency, Isolation, Durability* compliant database. These four properties of a transaction ensure database robustness over aborted or (partially) failed transactions.

[9] see `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`

Future support for OWL reasoning in Sesame could consist of implementing a SAIL that acts as a client on a DIG interface, and communicates with a DL reasoner to obtain the necessary answers for OWL semantics.

A somewhat more humble form of OWL support could come in the form of specifying additional entailment rules that capture a subset of the OWL semantics. While OWL's complexity is too high to be fully captured using Horn rules, it is possible to identify subsets of OWL that can be supported in this fashion (see also [ter Horst, 2004, Grosof et al., 2003]).

5.6.4 Context Support

The notion of context in terms of RDF graphs can be very useful to control and manipulate large, heterogeneous sets of RDF data. Although not part of the specifications of RDF, many RDF systems provide some form of provenance/context support, where the source of a statement (i.e., the URI of the document from which the statement originally comes) is stored as a facet of each statement.

Although RDF's reification mechanism is conceptually able to encode such information, in practice this may not be desirable: reification adds four additional RDF statements for each reified triple. To use this mechanism on every triple would therefore lead to an unacceptable increase in the size of the repository.

A non-semantical explicit form of *context support*, where an additional property of each statement is automatically and efficiently stored in the backend (e.g. as an extra column in a database table) and exposed to the world *as if* it were a form of reification, is a desirable option to have in an RDF repository: it will enable the grouping of sets of statements according to source, timestamp, version or whatever other grouping characteristic the user finds convenient, conceptually staying within the RDF specifications by making this information queryable as if it were reification, but in the backend making sure that a more compact specialized form of storage is used.

5.7 Related Work

The Jena Semantic Web Toolkit [Carrol and McBride, 2001] is a Java library that offers functionality for manipulating RDF models. It offers a model-centric RDF API that is rich in operations. The main differences between Sesame and Jena lie in the fact that Jena is model-centric while Sesame focuses on storage and retrieval: the central object in Sesame's API is the repository, whereas in Jena it is the more abstract concept of an RDF model. Moreover, Sesame offers more advanced querying support than Jena does (it supports several expressive query languages whereas Jena only supports one, not very expressive, language). Additionaly Sesame has more focus on scalability and persistence, offering a wider choice of persistence backends. Jena on the other hand focuses on ease of deployment as a library. Both projects have grown closer together in terms of functionality and intended audience however, and it is likely that this trend will continue in the future.

KAON [Bozsak et al., 2002] is an ontology management platform designed by the university of Karlsruhe that targets business applications. Its feature set is overlapping to a large degree with that of Sesame, though it is a more comprehensive framework which offers not only storage middleware but also editors and other end-user tools. However, it lacks support for a declarative query language.

RDFSuite [Alexaki et al., 2000] is a database system for RDF and RDF Schema that implements the RQL query language. It focuses exclusively on fast storage and retrieval of RDFS data and as such has a narrower target audience and a more limited feature set than the Sesame framework.

Kowari [Wood et al., 2005] is a recent development effort. It is a triple store that uses a native storage and indexing scheme designed for maximising querying speed and scalability. Rather surprisingly its indexing scheme is based on binary trees rather than the more conventional B or B+ trees for fast disk lookups. It implements a single query language. Kowari focuses on a single type of storage and optimizes for that. A possible direction for future cooperation between the Kowari and Sesame projects could be to wrap Kowari as a SAIL implementation in the Sesame framework.

5.8 Discussion

In this chapter we have presented Sesame, a generic framework for storing and querying both RDF and RDFS information. Sesame is an important step beyond the previously available storage and query devices for RDF, since it is the first publicly available implementation of a query language that is aware of the RDFS semantics.

An important feature of the Sesame architecture is its abstraction from the details of any particular repository used for the actual storage. This makes it possible to port Sesame to a large variety of different repositories, including relational databases, RDF triple stores, and even remote storage services on the Web.

Sesame itself can be deployed both as a programming library or as a server-based application, and can therefore be used as a remote service for storing and querying data on the Semantic Web. As with the storage layer, Sesame abstracts from any particular communication protocol through its use of a generic Repository API, so that Sesame can easily be connected to different clients by writing differen implementations of this Repository API.

We have constructed several concrete implementations of the generic architecture, using relational databases (MySQL, PostgreSQL, Oracle, SQL Server), main memory storage or native disk storage and using HTTP and RMI as remote communication protocol handlers.

Important next steps to expand Sesame include implementing transaction rollback support, context support, versioning, extension from RDFS to OWL reasoning and implementations for different database systems. This last feature especially will be greatly facilitated by the fact that the current RDBMS SAIL implementation is a generic SQL99 implementation, rather than specific for a particular DBMS.

Chapter 6

Inferencing

In this chapter, we will look at several algorithms for supporting inferencing over RDF models. The presented algorithms have been implemented and tested in the Sesame framework presented in chapter 5. We will analyze space-time tradeoffs of different strategies and consider which algorithm is most suitable in which use case.

Parts of this chapter have been published in [Broekstra et al., 2002], [Broekstra and Kampman., 2003] and [Stuckenschmidt and Broekstra, 2005].

6.1 Introduction

As discussed in chapter 2, the Resource Description Framework (RDF) [Lassila and Swick, 1999] specifies a simple model for knowledge representation. RDF Schema (RDFS) [Brickley and Guha, 2000] adds additional expressive power and semantics to this basic model. The combined language has a simple first order predicate logic as its foundation, the semantics of which are described in [Hayes, 2004].

In this chapter, we will look at algorithms and implementation issues related to supporting RDF's semantics. We will present a simple pruning iterative forward chaining algorithm for computing the deductive closure of an RDF model and discuss its implementation in the Sesame framework. In later sections, we will look more closely at different inferencing strategies; we will analyze space-time tradeoffs in such strategies and propose a mix between offline (forward chaining) and online (backward chaining) reasoning.

The chapter is organized as follows. In section 6.2 we introduce the RDF Semantics specification and present a proof system that will be used as the basis for the inferencing strategies. In section 6.3, we briefly look at related work. In section 6.4, we present the Sesame algorithm, an pruning iterative forward chaining algorithm. Section 6.4.3 presents performance figures of the Sesame algorithm on several data sets. In section 6.5 we perform an in-depth analysis of the time-space tradeoffs involved in different reasoning strategies (such as full forward chaining vs. full backward chaining), and in section 6.6 we outline a novel approach for RDF reasoning that attempts to find an

equilibrium between minimal space complexity and maximum time efficiency. We give a proof of correctness and completeness of the algorithm in section 6.6.3. Finally, we present our conclusions in section 6.8.

6.2 The RDF Semantics

RDF models can be seen as a set of statements or as the graph induced by these statements. RDF Schema models are RDF models where a subset of the triples use a designated vocabulary with a special meaning defined in the RDF Semantics specification. The special meaning allows us to derive new statements. In the following, we briefly describe a proof system for RDF schema that has been proposed by [Gutierrez et al., 2004].

6.2.1 A Proof System

The RDF semantics specification [Hayes, 2004] does not only provide a model theoretic semantics for RDF and RDF schema, but also provides an alternative specification of the semantic in terms of a deduction system. The deduction system consists of a set of *axioms* (RDF statements) about the nature of RDF and RDF schema primitives. For example the fact that `rdf:type` is a property is asserted by the axiom `(rdf:type rdf:type rdf:Property)`. Furthermore the deduction system includes a set of *inference rules* that can be used to derive new statements from existing ones. We list these inference rules below[1], because we extensively refer to individual rules throughout the chapter. For the full set of axioms, we refer to the specification.

Rule 1 *Every predicate is of type Property*

$$\frac{(X\ A\ Y)}{(A\ \texttt{type}\ \texttt{Property})} \tag{1}$$

Rule 2 *if U is the subject of predicate A and the domain of A is X, then U is of type X*

$$\frac{(A\ \texttt{domain}\ X),(U\ A\ Y)}{(U\ \texttt{type}\ X)} \tag{2}$$

Rule 3 *If V is the object of predicate A and the range of A is X, then V is of type X*

$$\frac{(A\ \texttt{range} X),(Y\ A\ V)}{(V\ \texttt{type}\ X)} \tag{3}$$

Rule 4 *Every U used as a subject or an object is of type Resource*

$$\begin{array}{c}\frac{(U\ A\ B)}{(U\ \texttt{type}\ \texttt{Resource})} \\ \frac{(B\ A\ U)}{(U\ \texttt{type}\ \texttt{Resource})}\end{array} \tag{4}$$

[1] For reasons of brevity we deliberately omit rules concerned with the treatment of literals. All results in this chapter are also valid if we include them.

Rule 5 *if U is a subproperty of V and V of X then U is a subproperty of X (transitivity)*

$$\frac{(U \texttt{ subPropertyOf } V),(V \texttt{ subPropertyOf } X)}{(U \texttt{ subPropertyOf } X)} \tag{5}$$

Rule 6 *Every property is a subProperty of itself (reflexivity)*

$$\frac{(U \texttt{ type Property})}{(U \texttt{ subPropertyOf } U)} \tag{6}$$

Rule 7 *If A is a subproperty of B then for all U and Y which are connected by a relation A it holds that they also are connected by a relation B.*

$$\frac{(A \texttt{ subPropertyOf } B),(U\ A\ Y)}{(U\ B\ Y)} \tag{7}$$

Rule 8 *All classes are a subclass of the class Resource*

$$\frac{(U \texttt{ type Class})}{(U \texttt{subClassOf Resource})} \tag{8}$$

Rule 9 *If U is a subclass of X and V is an instance of U then V is also an instance of X (instance inheritance)*

$$\frac{(U \texttt{ subClassOf } X),(V \texttt{ type } U)}{(V \texttt{ type } X)} \tag{9}$$

Rule 10 *Every class is a subclass of itself (reflexivity)*

$$\frac{(U \texttt{ type Class})}{(U \texttt{ subClassOf } U)} \tag{10}$$

Rule 11 *If U is a subclass of V and V of X then U is a subclass of X (transitivity)*

$$\frac{(U \texttt{ subClassOf } V),(V \texttt{ subClassOf } X)}{(U \texttt{ subClassOf } X)} \tag{11}$$

Rule 12 *All ContainerMembershipProperties are a subproperty of member.*

$$\frac{(U \texttt{ type Containermembershipoperty})}{(U \texttt{ subPropertyOf member})} \tag{12}$$

Rule 13 *all datatypes are subclasses of Literal.*

$$\frac{(U \texttt{ type Datatype})}{(U \texttt{ subClassOf Literal})} \tag{13}$$

The semantics specification also provides a proof that the proof system corresponds to the model-theoretic semantics in the sense that the set of all statements that can be derived by iteratively applying the inference rules are exactly those statements that follow from the model-theoretic semantics.

6.2.2 Closure

When talking about an RDF model, we have to distinguish between the statements that are explicitly contained in the model and the information that is implicitly contained and can be made explicit by applying the rules specified above. We call the set of all statements – explicit or implicit – the closure of a model.

Definition 4 (Closure) *The closure of an RDF graph G is the graph defined by the set of all triples that are implied by G. We denote the closure of a graph G as* $c(G)$.

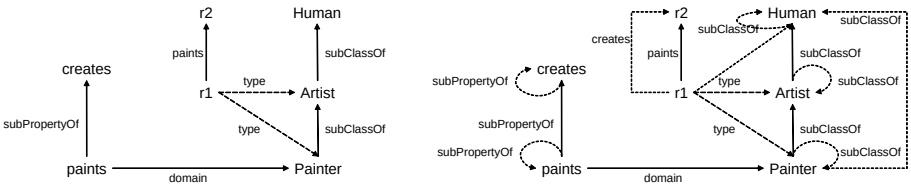
We can observe several properties of the closure:

Lemma 1 *The relation between a graph G and its closure is as follows:* $G \subseteq c(G)$.

This trivially follows directly from the definition of closure. We will use this property in the analysis of space requirements in section 6.5.1.

Lemma 2 (Closure Uniqueness) $c(G)$ *is unique for any graph G.*

We consider this trivially true by the definition of closure: given a fixed set of inference rules that is proven closed and a fixed set of initial facts, the deductive closure is uniquely determined.



(a) Original Model (b) Closure

Figure 6.1: Closure of a Model

Figure 6.1 illustrates the notion of closure using a simple example.

6.3 Related Work

Since the first publication of the model theoretic semantics of RDF schema, there has been an interest in the use of the inference rules provided by the specification to provide efficient reasoning services. Lassila [Lassila, 2002] reports experiments in using the inference rules specified in the RDF semantics for RDF schema reasoning. He presents a number of rewrite patterns that let a query engine access an RDF model as if it was the closure. The approach, however, is limited to a subset of the RDF schema semantics. In particular, inferences using domain and range restrictions are not completely captured. The SWI-PROLOG semantic library [Wielemaker et al., 2003], a PROLOG based RDF

infrastructure also provides limited support for RDF schema reasoning. In particular, the transitive closure of subclass and subproperty relations can directly be accessed, whereas other inferences have to be implemented as a backward inference engine in PROLOG. A similar approach is presented by [Christophides et al., 2003] who propose to create special index structures for answering queries about hierarchical relations in RDF models. Queries about these relations are reformulated to use the index values instead of the resource names. These indices are created offline and have the same effect as a precomputation of the transitive closure. A thorough comparison between our approach and index construction remains work to be done.

6.4 Forward Chaining Entailment

Traditionally, forward chaining inference systems use the well-known Rete algorithm [Forgy, 1982] for efficient handling of reasoning. An attractive property of this algorithm is that its performance is independent of the number of rules.

However, for our purposes, the Rete algorithm is less efficient: Rete requires the construction of a large network of nodes (the Rete) that caches facts from the knowledge base. On large datasets, the algorithm can run into serious memory/performance problems. Moreover, since we are dealing with a very specific proof system with a fixed number of entailment rules, the potential disadvantage of an algorithm that is performance-dependent of the number of rules is not particularly relevant. In fact, we suspect that an algorithm specifically optimized to this fixed set of rules will outperform a generic approach like Rete.

In this section, we present an iterative, pruning forward chaining algorithm for RDF entailment. The main benefits to this approach are ease of implementation, low memory footprint, and fast query answering – since reasoning is done offline, query answering only needs to do simple lookups. Since this algorithm has been implemented in the Sesame framework, we will refer to it as the Sesame algorithm.

A transaction T is a set of operations on an RDF model M that includes addition of RDF statements. The set N is the set of new statements asserted into M during T. We also define a set N_i as the set of statements inferred during iteration i.

Table 6.1 outlines the general form of the algorithm. It consists of a simple loop that iterates over the set of entailment rules and terminates when no new statements have been derived in the last iteration. It only applies a particular entailment rule in iteration i when this rule has been *triggered*: a fact has been newly derived in iteration $i-1$ that matches a premise of the entailment rule.

The algorithm is guaranteed to terminate: each new iteration is applied only to statements newly derived in the previous iteration. Since the total set of statements in the closure is finite, algorithm terminates when no new statements can be derived (that is, when the complete closure has been computed).

The application of the entailment rules is hardcoded into the algorithm itself. In table 6.2, the application of entailment rules 1 and 2 is shown as an example. As can be seen, the algorithm loops over the set of statements that was newly inferred during the previous iteration. Application of rule 1 is trivial since every statement s always matches

```
let N_0 = N
let i = 0
while (N_i ≠ ∅):
      let i = i + 1
      for each rule r:
         if (triggered(r)) then
            res_r ← applyRule(r, N_{i-1})
            add res_r to N_i
         endif
      endfor
endwhile
```

Table 6.1: Iterative pruning forward chaining algorithm

its premise. Therefore, the implementation of rule 1 simply loops over new statements and asserts that their predicates are of type `rdf:Property`.

Rule 2 is a more interesting case, since it has a more specific premise, and has in fact two premises that have to correspond at some point. As can be seen, the algorithm treats each premise match as a specific case (2_1 and 2_2, respectively). The algorithm checks for each statement $s \in N$ whether it matches one of the premises, and if it does, a matching statement for the other premise is found in M. Finding this match $m \in M$ is potentially very expensive: in a naive implementation it might require a linear search over the entire RDF model. However, note that every time such a search over M occurs, there are constraining factors. Smart indexing over the subject, predicate and/or object in the implementation of the algorithm can therefore significantly reduce this problem.

6.4.1 Trigger Optimization

In table 6.1 we have seen that application of an entailment rule is dependent on whether or not the rule is *triggered* by the previous iteration. The idea is that if in iteration $i-1$ an entailment rule produced a new statement that could be used as a premise for rule n in iteration i, rule n is triggered. We will refer to such a trigger as a *dependency* between two rules.

The dependencies between rules can be identified by a simple analysis. For example, the consequence of rule 1 is a statement of the form `(X rdf:type rdf:Property)`. Rules 2, 3, 4a, 6, 7 and 9 have premises that match with this statement pattern. In the case of rule 6, the match is specific (its premise is also `(X rdf:type rdf:Property)`), but for example in the case of rule 2, the match is very general: rule 2 has a premise `(X Y Z)` that by definition matches any statement.

Beyond this basic analysis of dependencies, however, we can make several more sophisticated optimizations to further prune the search space of the algorithm.

In table 6.3, we see an overview of the optimized dependencies between the entailment rules. We can immediately make a number of observations:

```
applyRule(i, N) :
      let N' = ∅
      if i = 1 then // rule 1
         for each s ∈ N do:
            create new statement t:
               (s.predicate rdf:type rdf:Property)
            add t to M
            add t to N'
         endfor
      else if i = 2_1 then // rule 2 first premise
         for each s ∈ N do:
            if s.predicate =rdfs:domain then:
               find statement m ∈ M:   m.predicate = s.subject
               create new statement t:
                  (s.subject rdf:type m.object)
               add t to M
               add t to N'
            endif
         endfor
      else if i = 2_2 then // rule 2 second premise
         for each s ∈ N do:
            find statement m ∈ M:
               m.predicate =rdfs:domain∧m.subject = s.predicate
            create new statement t:
               (m.subject rdf:type s.object)
            add t to M
            add t to N'
         endfor
      endif // (etc.  for other rules)
      return N'
```

Table 6.2: Application of entailment rules in the Sesame algorithm

- for every entailment rule with two premises, there are two entries in the table (indicated with a subscript index).
- despite the fact that rules 1, 4a and 4b have a very general premise pattern, there are almost no dependencies listed for these rules.

The duplication of rules with two premises is an optimization that has to do with the fact that each rule iterates over the set N_{i-1}, that is, the set of statements inferred in the previous iteration. As we have seen in table 6.2, each rule with two premises is treated separately by the algorithm. We have also seen that for a rule with two premises,

	triggers rule:									
rule:	1	2_1	2_2	3_1	3_2	4a	4b	5_1	5_2	6
1		•		•		•				•
2_1		•		•						•
2_2		•		•						•
3_1		•		•						•
3_2		•		•						•
4a		•		•						
4b		•		•						
5_1								•	•	
5_2								•	•	
6		•		•						
7_1		•	•	•	•			•	•	•
7_2		•	•	•	•			•	•	•
8		•		•						
9_1				•						•
9_2				•						•
10		•		•						
11_1										
11_2										
12		•		•			•	•	•	
13		•		•						

	triggers rule:									
rule:	7_1	7_2	8	9_1	9_2	10	11_1	11_2	12	13
1	•				•					
2_1	•		•		•	•			•	•
2_2	•		•		•	•			•	•
3_1	•		•		•	•			•	•
3_2	•		•		•	•			•	•
4a	•				•					
4b	•				•					
5_1	•	•								
5_2	•	•								
6	•									
7_1	•	•	•	•	•	•	•	•	•	•
7_2	•	•	•	•	•	•	•	•	•	•
8	•			•			•	•		
9_1	•		•		•	•			•	•
9_2	•		•		•	•			•	•
10	•									
11_1	•			•			•	•		
11_2	•			•			•	•		
12	•	•								
13	•			•			•	•		

Table 6.3: Optimized dependencies between RDFS entailment rules

a potentially expensive search over the entire model M is invoked. By separating the premises (and the entailment rules) in this fashion, we eliminate half these searches.

Another optimization made to the dependency table is that for rule 1, no triggers are

defined, despite the fact that its premise pattern matches virtually every entailment rule consequent. The justification for this optimization lies in the nature of rule 1: it identifies new predicate nodes. However, the entailment rules (except rule 7) never assert new predicates (observe that apart from rule 7, in the consequent part of the rules, a variable never occurs in the predicate). The reason rule 7 does not trigger rule 1 is somewhat more complex and has to do with the axioms present in the proof system, specifically the axiomatic statement `(rdfs:subPropertyOf rdfs:range rdf:Property)`. If we apply this axiom, in combination with the consequent of rule 7, to rule 3 (which is triggered by rule 7), rule 3 derives that the predicate that rule 7 introduced is of type `rdf:Property`. Therefore, after the first iteration (in which all rules are applied), there is by definition nothing left for rule 1 to entail (because either it has already been entailed, or rule 3 in combination with the aforementioned axiom takes care of it), and it can be safely pruned from further iterations. Similar optimizations have been made for rules 4a and 4b: these rules assert that the subject and object are of type `rdfs:Resource`, respectively. However, the entailment rules do not introduce new resources, only new statements built from existing resources. Therefore, after the first iteration, rules 4a and 4b can be safely pruned from further iterations.

6.4.2 Correctness and Completeness

The Sesame algorithm iteratively applies the entailment rules of the proof system, as presented in [Hayes, 2004]. In the semantics specification, the correctness of this approach is proven.

The specification also proves that exhaustive forward chaining is complete. In the previous section, we have shown how pruning reduces the search space without affecting completeness of the approach.

6.4.3 Performance

The Sesame algorithm has been implemented in the Sesame framework. In this section we present a few performance figures obtained with an early implementation of the presented algorithm. We used a number of different data sets in testing our approach.

- **OpenWine** is an open source data set that contains information about different wines. It is available from `http://www.openwine.org/`.
- **SUO** stands for "Standard Upper Ontology" and is a DAML+OIL representation of the SUO IEEE effort to standardize an upper ontology[2]. The DAML+OIL file is available from `http://www.daml.org/ontologies/uri.html`.
- **CIA** is an RDF representation of the CIA World Factbook. It is an enhanced version of the RDF representation produced by the On-To-Knowledge IST project[3] and is available from `http://www.openrdf.org/rdf/CIA/`.

[2] See `http://suo.ieee.org/`

[3] see `http://www.ontoknowledge.org/`

data set	explicit	closure	increase
OpenWine	4310	5289	23%
CIA	26285	30260	15%
SUO	4071	12498	206%
Wordnet	273681	373485	37%

Table 6.4: Increase in number of statements through closure computing

dataset	closure computing (norm)
CIA	1.45
SUO	1.41
Wordnet	1.13

Table 6.5: Performance overhead of closure computing, normalized against simple upload

- **Wordnet** is a data set containing the Wordnet 1.6 schema (wordnet-20000620.rdfs) and the set of nouns (wordnet_nouns-20010201.rdf). These files are available for download at `http://www.semanticweb.org/library`.

A potential problem with forward chaining closure computing is the cost of storage space. Indeed, for many large or complex models the size of the closure may be such that the approach becomes impractical (we will address this problem in section 6.5 and further). However, as can be seen, the increase in number of statements through closure computing varies per data set. In most of our test cases it is well under control and does not exceed 50%. The only exceptions to this rule is the SUO data set. This large increase is caused by the fact that it consists exclusively of a large class hierarchy that is both broad and deep: since virtually every statement in the set is a schema statement a lot of inferencing rules are applicable to all statements.

The overhead in storage space for the complete closure of the model for these data sets introduced is shown in table 6.4.

In table 6.5 we see the comparative performance of closure computing using the Sesame algorithm (as implemented in Sesame release 0.8), normalized against simple upload without any form of closure computing.

We can observe from these result that using the Sesame algorithm for closure computing adds an overhead, but that this overhead is not excessive.

6.5 Space-Time Tradeoffs

As we have seen in the previous sections, forward chaining closure computation (using an iterative algorithm like Sesame), has advantages in terms of query answering performance and run-time memory requirements. However, there is a trade-off between run-time complexity and the amount of space needed to store the deductive closure. In

the second part of this chapter we analyze these space requirements of computing the deductive closure using a number of large real-life RDF models and compare it to the minimal space needed for storing the model.

In a recent study Guo et al. revealed the limitations of current systems with respect to handling large amounts of data both in terms of upload and query time [Guo et al., 2004]. In the following sections, we propose a novel strategy for RDF reasoning that combines offline computation based on an extensional semantics for RDF schema with a simple form of online reasoning. We prove the completeness and correctness of our reasoning method and evaluate our method with respect to space requirements and run-time behavior.

6.5.1 Analysis of Space Requirements

The definition of closure only allows us to make qualitative assertions about its relative size. The actual ratios will vary significantly based on the nature of the model (in the extreme case, all models will be of the same size). In order to be able to better judge the impact of closure computation in terms of space requirements compared to the actual representation we performed a number of experiments with real life data. We used the Sesame framework [Broekstra et al., 2002], release 1.0, for performing a number of reasoning experiments on realistic data sets. The system uses the forward chaining algorithm introduced in section 6.4. The algorithm is applied when RDF data is uploaded [Broekstra and Kampman., 2003]. Besides using the standard rule set mentioned above, the system can also perform reasoning using different sets of inference rules specified in a specific format. We use this feature later to implement an alternative reasoning approach.

6.5.2 The Test Setup

In our analysis we used existing RDF data sets that contain a schema as well as instance information. From the available options we chose four models two of which mostly contain instance information and two that mainly contain schema information.

CIA World Fact Book The CIA World Fact Book model contains information about the countries of the World. The model has a small schema that mainly describes the structure of the information, the majority of the statements describe properties of the different country resources. The version of the model we used contains the information from the 1999 fact book with extended schema definitions. It contains about 26.000 statements about almost 1.000 resources.

TAP KB The TAP KB is a collection of data sets about different areas of daily life. It contains information about products, geographic places, organizations, famous athletes and musicians. Each of the these areas comes with a small schema that again mostly defines the data structure. We used the individually published subareas of TAP KB mentioned above, because the complete model turned out not to be legal RDF. The model in the experience contains a bit more than 100.000 statements about more than 35.000 resources.

Wordnet Wordnet is a lexical resource that defines terms as well as their descriptions and semantic relations between them. Most of the contents of WordNet, i.e. information about nouns, their descriptions, hyponym and similarity relations is published in RDF. The published files do not make use of RDF schema vocabulary and therefore to not allow meaningful reasoning in the term hierarchy. In order to overcome this problem, we added two statements declaring nouns to be classes and hyponymy to be a subproperty of the subclass relation. The model we used in the experiment contained almost half a million statements about almost 100.000 resources.

Teknowledge Ontologies The company Teknowledge provides a set of ontologies among them the Standard Upper Model (SUMO) as well as the Mid-level Ontology (MILO) a general refinements along with a number of domain ontologies about different areas of interest. In contrast to the models described above, these models contain mostly schema information. In the case of the Teknowledge Ontologies, this schema information is organized in different metalayers making classes and properties instances of higher level concepts. For this experiment, we used OWL versions of SUMO, MILO and domain ontologies about Geography, Government, Economy and Transportation as found in the DAML ontology library. The overall model contained almost 20.000 statements about more than 5.000 resources.

For each of these datasets we analyzed the distribution of different types of statements, in particular schema statements as indicated by the properties `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range`. In particular, we were interested in how the number of these statements change when computing the closure in order to get an impression of what kinds of statements have the most impact on the growth of the model and therefore have potential for reducing the space requirements. In a second step, we also analyzed the involvement of the different inference rules explained in section 6.2. From this analysis we hope to get an inside in the origin of the new statements that are added to the closure. We used the Sesame system [Broekstra et al., 2002]. The system can be configured to automatically compute the closure of an RDF model during upload by applying exhaustive forward chaining. We used the RDF API of Sesame to count ground and inferred statements. Information about the application of the different inference rules can be obtained from Sesame's system log. In the following section, we summarize and discuss the results of these analyzes.

6.5.3 Results

The results of our experiments are summarized in figures 6.2 and 6.3. Figure 6.2 compares the distribution of different types of statements in the original models and in the closure. When we look at the statistics for the original files we see that the first two models almost only consists of type and non-schema statements reflecting the absence of a sophisticated model. The later two models also contain a significant amount of subclass relations. Only the Teknowledge model also contains a noticable number of subproperty relations and domain and range restrictions.

Not surprisingly the differences in the extent of the available schema has a major impact on the distribution of types of statements in the closure. We see that for the first

	type	subClassOf	overall
CIA Fact Book	11.03	6.67	1.15
TAP KB	5.19	5.27	2.24
Teknowlegde	2.71	4.86	2.47
Wordnet	2.78	7.73	2.66
Average	6.13	6.13	2.18

Table 6.6: Increase of statements (factor)

two models, which do not contain extensive schema information, the type of statements that increase are the type statements. For the models with a richer model, the main increase can be observed in the amount of subclass relations. Table 6.6 summarizes the degree of increase for these types of statements. We see that while the average increase of the model is by a factor of 2.18, the average increase of type statements is 6.13, the one for subclass statements is 6.6.

Looking at the use of the different inference rules, we see that the extent of the schema information has a significant impact on the kinds of rules that are used to derive new statements. For the first two data sets, we see that almost only rules that derive type statements are applied. In particular, these are rules 2, 3, 4 and 9 (compare figure 6.3). In the presence of richer schema information like in the case of the latter two models, much more rules become relevant in closure computation. We can see that rules 2,3 and 9 still play a role. In addition, rules that derive subclass statements become important. In particular rules 7,8,10 and 11 play a significant role in computing the closure.

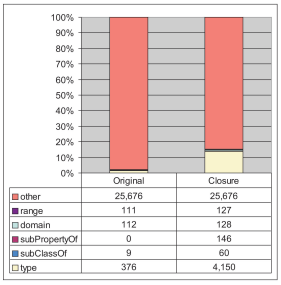
6.5.4 Conclusions

Coming back to our questions of how we can reduce the space requirements while still allowing for efficient schema-aware querying at runtime, we can draw a number of conclusions from the results of the experiments. The main conclusion is that type and subclass relations offer the greatest potential for reducing the size of the model to be stored if they are not computed offline, but are derived at runtime. This can be done by excluding the corresponding rules that we identified in the second part of the experiment from the offline computation step.

Considering the nature of the different relations, we argue that only excluding type statements from the closure is a good approach for achieving a hybrid reasoner. The reason for this is the following:

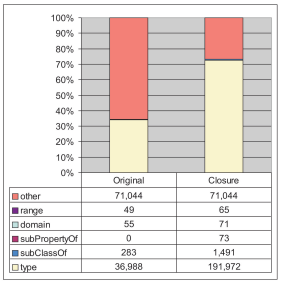
1. The type relation shows a significant increase in all models, regardless of whether the model has an extensive schema or not.

2. Computing the transitive closure of the subclass hierarchy is a major part of closure computation as the derivation of other statements, in particular type statements,



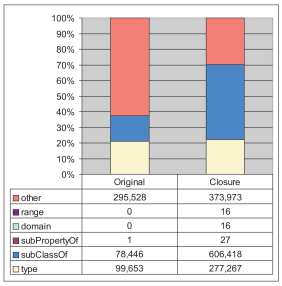
	Original	Closure
other	25,676	25,676
range	111	127
domain	112	128
subPropertyOf	0	146
subClassOf	9	60
type	376	4,150

(a) CIA Fact Book



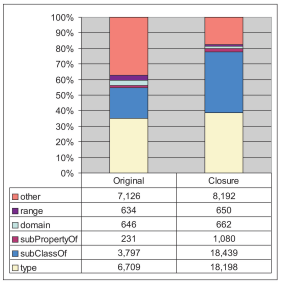
	Original	Closure
other	71,044	71,044
range	49	65
domain	55	71
subPropertyOf	0	73
subClassOf	283	1,491
type	36,988	191,972

(b) TAP KB



	Original	Closure
other	295,528	373,973
range	0	16
domain	0	16
subPropertyOf	1	27
subClassOf	78,446	606,418
type	99,653	277,267

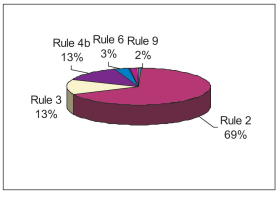
(c) Wordnet



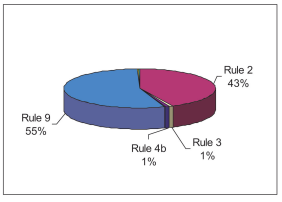
	Original	Closure
other	7,126	8,192
range	634	650
domain	646	662
subPropertyOf	231	1,080
subClassOf	3,797	18,439
type	6,709	18,198

(d) Teknowledge

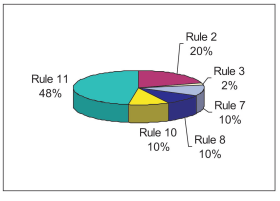
Figure 6.2: Types of Statements



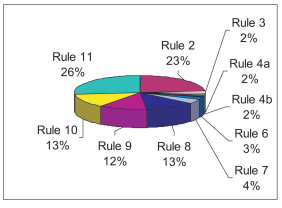
(a) CIA Fact Book



(b) TAP KB



(c) Wordnet



(d) Teknowledge

Figure 6.3: Percentage of Statements Inferred per Inference Rule

relies on it. We can assume that deriving type statements can be done with almost no overhead if the closure of the subclass relation has already been computed.

3. As updating the closure in the face of changing information is a major bottleneck of the approach, it is preferable to exclude information from the stored closure that is likely to change. In the case of RDF, it is clear that the schema part of a model is more stable than the instance information. Therefore the set of type statements will typically change more often than the set of subclass statements when instances are added or deleted.

In summary, we conclude that a good hybrid approach is excluding redundant type statements – type statements that can be inferred from the original model using the deduction system in section 6.2 – and computing these at query time. This goal can be achieved by delaying the application of the corresponding inference (in particular rules 2,3, 4 and 9) and performing reasoning at run-time instead.

6.6 Schema Closure: A Novel Approach

In this section, we provide details of the approach sketched at the end of the previous section. In particular, we define a set of inference rules for inferring only schema statement off-line. Further, we present an algorithm for schema aware query answering that is based on query rewriting techniques for answering queries about facts that are not contained in the explicit statements. In the last part of this section we analyze correctness and completeness of the method wrt. the original rule set.

6.6.1 A Proof System for Schema Closure

The goal of the proof system is to infer all derivable statements from an RDF model except for implicit type statements. For this purpose we first include all rules that do not create new type statements. These are rules 5, 6, 8, 10, 11, 12 and 13 from the original proof system (section 6.2). Just using this reduced set of rules is problematic, because some rules of this set have type statements in the rule body (rules 6, 8, 10, 12 and 13). This means that we will also lose inferences at the schema level if we just omit the other rules. Instead we take an approach where we restrict the application of general type inference rules to cases where they compute certain type statements that are potentially input to other rules.

We have to pay special attention to rule 7, because it potentially derives any kind of statement. There are different options of dealing with this situation:

- We can include rule 7 into the set of rules for off-line computation. This guarantees that we do not miss any schema statements, but also means that we might compute some undesired type relations.

- We can restrict the use of rule 7 to cases where the derived statements is a potential input for one of the other rules. This guarantees that we do not generate any

unwanted statements in the off-line phase, but it also means that we might miss statements about non-schema elements.

In our approach we decided to choose the first option. This leads to slightly larger models, but reduces the complexity of online reasoning. The reason for choosing to include rule 7 is that in many cases, it does not play a significant role in the closure computation and will therefore not lead to a major increase in the model size.

Rule Instantiation

We restrict the general type inference rules by replacing them with a set of instantiated rules. In these rules, some of the variables have been replaced by the names of schema elements.

Rule 1: For the case of rule 1 this is not necessary, because it only computes type statements with respect to the class 'Property'. As these statements are input to rule 6, we leave rule 1 unchanged.

Rule 2: This rule compute general type statements based on a combination of domain definitions. We instantiate the variable X by schema elements that occur in the bodies of other inference rules, in particular `Class`, `Property`, `Datatype` and `ContainerMembershipProperty`. This leads to a set of four rules that replace rule 2 in the proof system:

$$\frac{(A\ \texttt{domain Class}),(U\ A\ Y)}{(U\ \texttt{type Class})}$$
$$\frac{(A\ \texttt{domain Property}),(U\ A\ Y)}{(U\ \texttt{type Property})}$$
$$\frac{(A\ \texttt{domain Datatype}),(U\ A\ Y)}{(U\ \texttt{type Datatype})}$$
$$\frac{(A\ \texttt{domain ContainerMembershipProperty}),(U\ A\ Y)}{(U\ \texttt{type ContainerMembershipProperty})} \tag{14}$$

Rule 3: This rule is equivalent to rule 2 except that it uses range definitions for inferring general type statements. Consequently, we instantiate rule 3 in the same way as rule 2. This leads to a set of four rules that replace rule 3 in our proof system:

$$\frac{(A\ \texttt{range Class}),(Y\ A\ V)}{(V\ \texttt{type Class})}$$
$$\frac{(A\ \texttt{range Property}),(Y\ A\ V)}{(V\ \texttt{type Property})}$$
$$\frac{(A\ \texttt{range Datatype}),(Y\ A\ V)}{(V\ \texttt{type Datatype})}$$
$$\frac{(A\ \texttt{range ContainerMembershipProperty}),(Y\ A\ V)}{(V\ \texttt{type ContainerMembershipProperty})} \tag{15}$$

Rule(s) 4: These rules are somewhat trivial as they only create statements indicating that the subject and the object of a statement are of type resource. As these statements

are not further used in any rule we omit these rules.

Rule 9: This rule computes new type relations based on existing ones and information about subclasses. We can restrict it to schema elements by instantiating it in the same way as rule 2 and 3. We replace rule 9 by the following set of instantiated inference rules:

$$\frac{(U \texttt{ subClassOf Class}),(V \texttt{ type } U)}{(V \texttt{ type Class})}$$
$$\frac{(U \texttt{ subClassOf Property}),(V \texttt{ type } U)}{(V \texttt{ type Property})}$$
$$\frac{(U \texttt{ subClassOf Datatype}),(V \texttt{ type } U)}{(V \texttt{ type Datatype})}$$
$$\frac{(U \texttt{ subClassOf ContainerMembershipProperty}),(V \texttt{ type } U)}{(V \texttt{ type ContainerMembershipProperty})} \quad (16)$$

Schema Closure

The modifications to the original set of inference rules described above provides us with a proof system that defines a subset of the closure of a model that that contains the complete schema information but only a subset of the instance data. We denote this subset as 'schema closure' and formally define it in the following way:

Definition 5 *Let S be a deduction system consisting of rules 1, 5-8, 10-13, 14-16. The schema closure of a model G is the maximal model G' such that $G \vdash_S G'$. We denote the schema closure of an RDF graph as sc(G).*

The schema closure is created by applying a rule set that contains a subset of the original inference rules or refinements of original rules. Based on this fact, we can make the following assertion about the relation between the schema closure and other concepts related to RDF inference discussed in this chapter.

$$r(G) \subseteq G \subseteq sc(G) \subseteq c(G)$$

Our previous experiments suggested that computing the schema closure off-line and storing it instead of the original model, and using online reasoning to bridge the gap between schema closure and closure of a model, constitutes a near-optimal trade-off between space requirements for storing the model and time complexity of computing the closure. Before we test this claim in a second set of experiments, we draw our attention to the online-reasoning part.

6.6.2 Online Reasoning by Query Rewriting

The price of reducing the size of the model to store is that we still have to perform parts of the reasoning online. It turns out however, that the way we determined the statements to be pre-computed allows us to simply expand queries posed against the schema closure by replacing those parts of the query that asks for instance level statements by a more

complex query expression (of a fixed size !) that involves some schema elements. This means that we do not need an inference engine to provide schema-aware querying support as the same functionality can be achieved by a query-preprocessor expanding the query and sending it on to any RDF query engine. In the following, we describe the query expansion strategy and show that it is correct and complete.

Query Rewriting

The idea of using query rewriting to answer queries is not new. It has been extensively used in the database area for view-based information integration [Ullman, 1997]. The idea is that instead of answering a query directly the system derives a query that matches the data. The connection between the posed query and the query that is actually answered is defined by a view definition. Such views can be defined in two ways, global-as-view and local-as-view. In the global-as-view approach, elements from the query to be answered are determined in terms of a query over the data model. The local-as-view approach inverses this relation and defines elements of the actual data model in terms of a queries over the elements of the query schema [Halevy, 2001]. Despite having a higher computational complexity, the local-as-view is the approach of choice for information integration because it provides more flexibility for adding new sources. For our purpose, we can use a global-as-view approach as we do not deal with multiple information sources (the schema closure of the RDF model is our only source) and the relation between the models is always the same. Being able to use this approach, we benefit from the straightforward way, rewriting can be done in the global-as-view approach which basically consists of replacing all occurrences of the goal views goal predicate in the query by the body of the view definition.

We assume a query language that uses sets of triple patterns to define the subset of the RDF model to be selected. This assumption does not limit our approach as many widely used RDF query languages such as RDQL and SeRQL are based on this paradigm. In order to completely answer queries in such a language, we have to provide special mechanisms for triple patterns that contain a type relation.

These patterns potentially match statements that were not included in the schema closure but can be derived using the RDF semantics. The fact that the schema closure contains all schema related statements, we can use special view definitions to replace these patterns by extended ones that used compiled schema information to also capture implicit statements. We use special kinds of view definitions where the head is a single triple pattern and the body is a set of triple patterns. When rewriting a query, we use these view definitions to replace triple patterns that correspond to the head of a view by the set of patterns in its body.

Let us first consider view definition for substituting type statements. The first set of view definitions we use is a direct counterpart of rules 4a and 4b.

$$(S \texttt{ type Resource}) \leftarrow \{(S\ X\ Y)\} \quad (17)$$
$$(S \texttt{ type Resource}) \leftarrow \{(X\ Y\ S)\} \quad (18)$$

The next view definition is a counterpart of rule 9. The difference to the direct use of straightforward use of the rule is the fact that the model we are querying is known to explicitly contain all subclass statements. This means that no further reasoning about the subclass hierarchy is required. The query will directly return all statements that can be inferred using rule 9.

$$\begin{aligned}(S \texttt{ type } O) \leftarrow\ & \{(O \texttt{ subClassOf } X),\\ & (S \texttt{ type } X)\}\end{aligned} \quad (19)$$

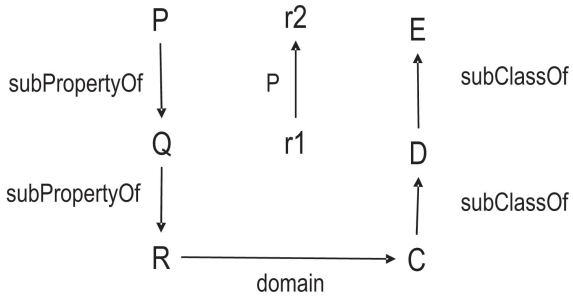


Figure 6.4: Complex Type inference

The most complex way of inferring type statements is in terms of a combination of subclass, subproperty and domain restrictions. We can find situations like the one shown in figure 6.4. Before being able to use rule 2 to infer that r1 is of type E, we first have to derive $(P \texttt{ subPropertyOf } R)$ in order to be able to apply rule 7. Afterwards, we have to derive $(C \texttt{ subClassOf } E)$ in order to establish the goal using rule 9. These intermediate facts, however, are already contained in the schema closure. We can use this fact to define the following view that explicitly contains the three statements necessary to establish the type relation.

$$\begin{aligned}(S \texttt{ type } E) \leftarrow\ & \{(C \texttt{ subClassOf } E),\\ & (P \texttt{ subPropertyOf } R),\\ & (R \texttt{ domain } C),\\ & (S\ P\ X)\}\end{aligned} \quad (20)$$

Note that this view definition is general enough to also capture the cases where the subclass and/or the subproperty relation does not have to be inferred but are contained in

the model. This case is covered, because the schema closure also contains the statements defining each class to be a subclass of itself and every property to be a subproperty of itself (compare rules 5 and 10). We can use the same idea to define a view definition that covers the derivation of type statements using a combination of subclass, subproperty and range statements.

$$\begin{aligned}(S\ \texttt{type}\ E) \quad \leftarrow \quad & \{(C\ \texttt{subClassOf}\ E),\\ & (P\ \texttt{subPropertyOf}\ R),\\ & (R\ \texttt{range}\ C),\\ & (X\ P\ S)\} \end{aligned} \tag{21}$$

Definition 6 (Query Rewriting) *Let V be a set of view definitions and G and RDF model. The rewriting approach to querying consists of the following steps:*

1. *detect all triple patterns matching definitions in V*
2. *for each pattern found, do*
   (a) *find all possible replacements of the pattern based on V*
   (b) *replace the triple pattern by a union query consisting of the replaced pattern and all possible replacements.*
3. *evaluate the query.*

The online part of our reasoning approach now consists of applying the rewriting method to the schema closure of an RDF model using the view definitions shown in equation 17 to 21.

6.6.3 Completeness and Correctness

In this section, we will discuss the formal properties of our reasoning approach on a slightly more formal level and give the central theorems establishing correctness and completeness of the approach. We will revisit the key ideas of the approach and present the proof idea without actually providing details of the proofs.

The Reasoning process designed above is a two-stage process. The first step is the computation of the schema closure. The second step is an algorithm for query rewriting that replaces simple queries by more complex ones that can be answered directly from the schema closure. The correctness and completeness of the second step strongly relies on some basic assumptions about the schema closure. We capture these assumptions by introducing two lemmata concerning the schema closure and use them to establish the main theorems.

The first lemma concerns the correctness of closure. This lemma basically claims that the proof system for the schema closure does not derive RDF statements that are not implied by the model.

Lemma 3 (Closure Correctness) *Let t be an arbitrary RDF statement and G a legal RDF model, then we have:*

$$t \in sc(G) \implies t \in c(G)$$

This lemma can easily be proven by looking at the set of rules we use to compute the schema closure. As mentioned before, the rules in this set are either also part of the original rule set that is known to be correct and complete or are special cases of such rules. In the proof we show this for each rule individually, the lemma directly follows by the monotonicity of RDF reasoning.

The second lemma we need states that the schema closure already contains all the subclass and subproperty statements as well as all type statements whose object is a schema element (i.e. a class, a property, a datatype or a container membership property).

Lemma 4 (Closure Completeness) *Let t = (s,p,o) be an arbitrary RDF statement with*

- $p \in \{\texttt{subClassOf}, \texttt{subPropertyOf}\}$ *or*
- $p = \texttt{type}$ *and* $o \in \{\texttt{Class}, \texttt{Property}, \texttt{Datatype}, \texttt{containerMembershipProperty}\}$

and G a legal RDF model, then we have:

$$s \in s(G) \implies s \in sc(G)$$

This lemma is slightly more difficult to prove. We have to show that there cannot be a statement of the above mentioned form that follows from the model and is not derived by our rule set. We can ensure this by looking at the different ways in which these statements can be derived using the original rule set. The different ways can be distinguished by the rule used to derive the statement. This rule provides us with information about other statements that are part of the model. We can now show that in each of these cases, there also is a rule that would derive the same statement in our rule set. The derivation of new facts is an iterative process, so we also have to consider cases where statements have been derived by chaining a number of rules. We can capture this in an induction step over the length of the derivation. We thereby first show that we cover all cases where only one rule is used and in the induction step show that for all of the relevant statements derived by the n+1st iteration of applying rules, there also is a way of deriving it using our rule set, provided that we could already derive all statements added in the previous iteration.

Using the results of the two lemmas, we can now state the main theorems about the correctness and completeness of our method. The first theorem establishes the correctness of the method by stating that every result that can be derived from the schema closure is indeed in the closure of the model.

Theorem 2 (Correctness) *Let V be a set containing the view definitions in equations 17 to 21 and* $rewrite_V(sc(G))$ *the set of all triples that can be derived from the schema closure of G using the rewriting approach, then*

$$t \in rewrite_V(sc(G)) \implies t \in c(G)$$

We can prove this theorem by looking at the set of view definitions. For each view, we can name a set of rules of the original rule set that, given the statements in the view body, derive the statement in the head of the view. As our method does not work on the original model, but on the schema closure, we need the result of lemma 1, which guarantees that all the statements in the closure can also be derived using the original rule set.

The second theorem establishes the completeness of our method by claiming that each statement in the closure of a model can also be derived by applying the rewriting approach to the schema closure.

Theorem 3 (Completeness) *Let V be a set containing the view definitions in equations 17 to 21 and* $rewrite_V(sc(G))$ *the set of all triples that can be derived from the schema closure of G using the rewriting approach, then*

$$t \in c(G) \implies t \in rewrite_V(sc(G))$$

We can prove this theorem by again looking at the different ways in which statements that are part of the closure can be derived using the original rule set. For some of these ways, we can directly see that this is the case, because the corresponding rule is also in the rule set we use for computing the schema closure. For the other cases, we can again use information from the premises of the rule used for deriving the statement to show that there is a view definition that can be used to derive the statement. Here, we make use of lemma 2, because the ability of the view definitions to derive a fact often relies on the assumption that all schema statements, in particular subclass and subproperty statements, are known. Again, we have to use an induction step to show that this result generalizes to statements that have been derived using a sequence of rules.

6.7 Experiments

The primary aim of our approach is to reduce the amount of data that has to be stored explicitly. By doing this we do not only hope to reduce the memory consumption but also to reduce upload and revision time for RDF schema models. In order to test whether our approach really achieves this goal, we carried out a second set of experiments using the Lehigh university benchmark [Guo et al., 2004]. This benchmark consists of a fixed schema about universities including aspects like departments, employees and courses. Furthermore, the benchmark has a statement generator that can be used to randomly produce instance data of arbitrary size. For our experiments, we created data for eight universities with an overall size of about one million statements. In the experiments

we uploaded this data to two setups of the Sesame framework. The first setup used the built-in inference engine that computes the complete closure of the model using the algorithm shown in section 6.4. In the second setup, we implemented our new rule set for computing the schema closure using Sesame's custom inferencer. Both experiments were carried out on a PC with an AMD Athlon 64 3000+ processor with a 2 Ghz CPU and 1 GB ram 512 MB of which were reserved as a heap for the java process. We used Java 2 version 1.4.2 and Sesame 1.1-RC2 with MySQL 4.0.21nt as a physical storage. The results from the experiments are discussed in the following.

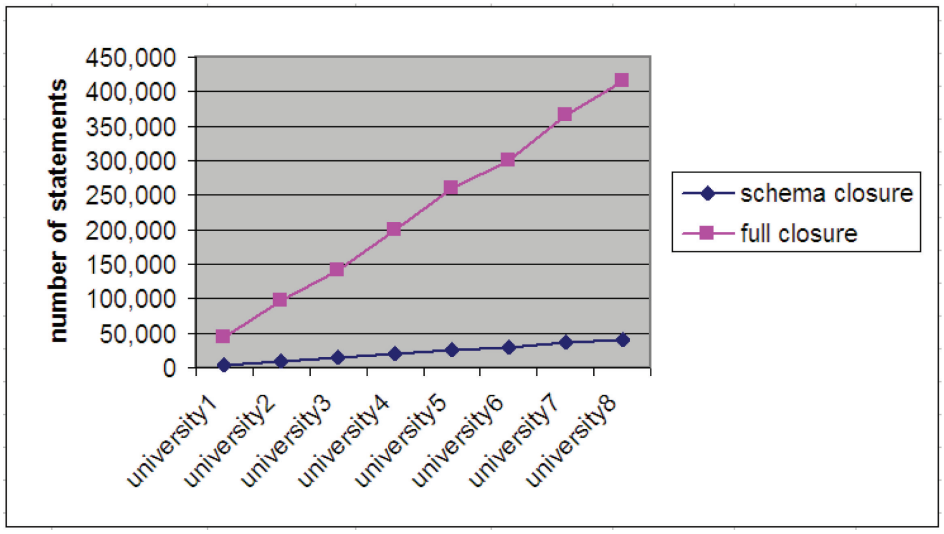


Figure 6.5: Growth of inferred statements

The first aspect we looked at is the size of the model in terms of number of statements. In particular, we were interested in the relation between the number of additional statements in the closure and in the schema closure, as well as the relation to the overall size of the model. Figure 6.5 shows the number of statements added. The experiment shows that the use of the schema closure instead of the full closure leads to significant reduction of the number of statements added. In the case of the full closure, the number of statements added is more than 40% of the number of explicit statements. In this test, this means that we have to deal with almost half a million additional statements. For the schema closure, the number of statements added is less than 5% of the number of original statements. This difference is quite significant as it amounts to a ration of 1:10 when comparing the size of the different closures. A question that remains to be answered is whether the reduction of additional statements is significant at all. Indeed figure 6.6 shows that explicit statements are still the dominant part of the model. We can also see, however, that size of the schema closure is almost the same, whereas the full closure is noticeably larger than the original model . As we have to store the explicit

facts anyways, computing the schema closure is an additional benefit that comes at low costs.

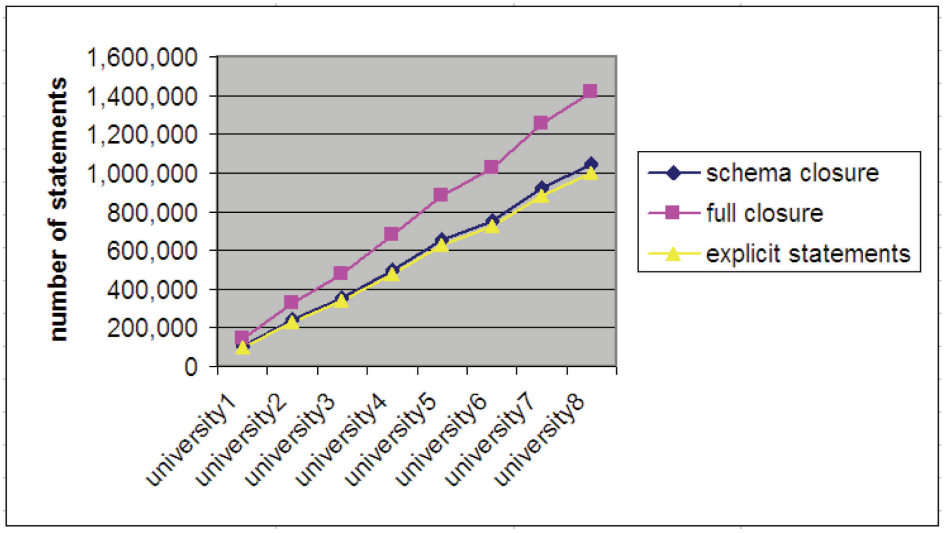


Figure 6.6: Growth of the overall model.

We also analyzed the impact of the reduced number of statements on the upload time for the model. The question was, whether the computation of the schema closure instead of the full closure significantly reduces upload time. Figure 6.7 shows the results of this experiment. We can see that the reduction in the size of the model also leads to a reduced upload time. In the case of larger models like the one used in the experiment, the difference is significant. In the experiment shown, an upload of the model with full closure computation took about 45 minutes whereas the upload of the model with a schema closure computation only took about 30 minutes. This is a reduction of the upload time of more than 30%.

We can expect that the reduction of the size of the closure has a similar effect on other management tasks. In particular, this will hold for the update of an RDF model. This is currently a very expensive task, because for each inferred statement we have to check whether it is still supported by the ground facts. We did not yet manage to run experiments to compare the time needed for updating the full closure versus updating the schema closure, but we expect that for this task the savings will even be larger, because we have to check less statements for validity and this check itself has to include less information.

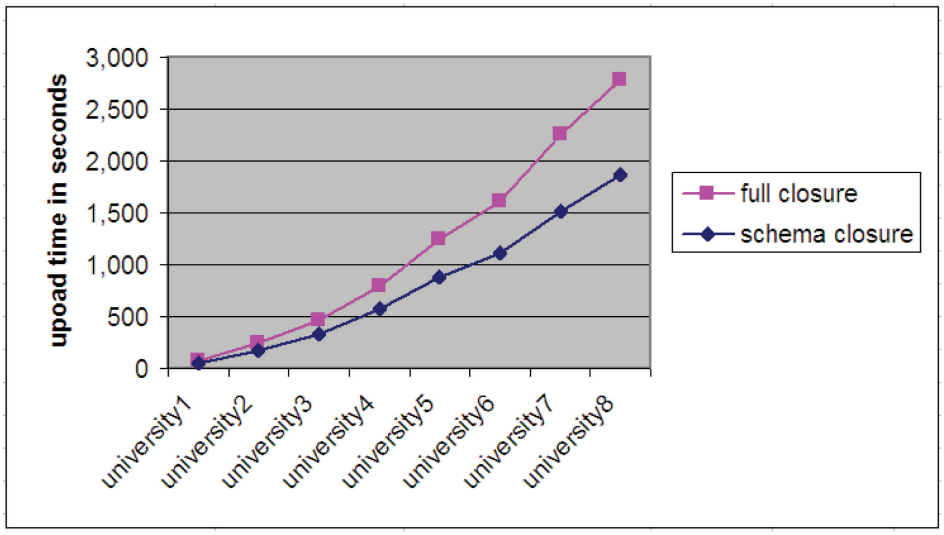


Figure 6.7: Upload time for the models

6.8 Discussion

In the first half of this chapter, we presented an iterative pruning forward chaining algorithm that does full closure computation for the RDF Semantics with good performance on medium-sized datasets. Its advantages are ease of implementation, low run-time memory requirements and good query performance.

In the second half, we presented an approach for RDF schema reasoning to support schema aware query answering that combines partial offline closure computation with view based query rewriting. The approach has the same advantage as a complete offline schema computation in the sense that an RDF query engine can be used to compute answers without further reasoning. On the other hand, the reduction of the closure to a subset of the complete closure reduces space requirements and upload time which is particularly significant when working with very large models. We expect similar improvements for other time consuming tasks such as the update of a model. For smaller models of computing the complete closure is still a valid approach. We implemented and tested our method on a benchmark dataset using the Sesame system.

In contrast to other rewriting-based approaches like [Lassila, 2002], we cover the complete RDF semantics by pre-computing the transitive closure of hierarchical relations. This pre-compilation is also provided for example by the SWI semantic web library [Wielemaker et al., 2003], but they do not provide methods for completing the closure in the online reasoning step. Both methods had problems in dealing with domain and range restrictions. Using the query rewriting strategy presented here, we can also cover these relations.

The approach that is probably closest to the work presented here is the Jena reasoning engine which combines forward and backward chaining. The main difference is that instead explicitly storing derived facts, Jena derives domain-specific rules to be used in the backward chaining step. This probably leads to an even more compact representation but requires the use of a rule engine to answer queries. A comparison of the two approaches would be an interesting exercise.

A potential drawback of the method is the time needed to answer rewritten queries that are more complex than the original query. In particular, being able to efficiently process these queries, the query engine has to be able to deal with union queries in an efficient way. As Sesame currently only provides very basic support for union queries without any optimization techniques, we were not able to provide meaningful figures about the efficiency of the query rewriting approach. Providing efficient methods for this problem is a topic of future research that can be based on existing work on database query optimization. As initial work on this approach was driven by our work on developing support for distributed RDF querying [Stuckenschmidt et al., 2004b], next steps will also include an investigation of how this method can help to provide reasoning support for schema-aware querying in a distributed setting.

Chapter 7

Truth Maintenance

In this chapter, we study a number of practical issues that arise when computing the deductive closure of an RDF Schema and dealing with the consequences of delete operations. We present a Truth Maintenance algorithm that makes use of dependencies between statements to deal with 'non-monotonous' updates (i.e. delete operations) to an RDF Schema knowledge base. We present how this algorithm has been implemented in the Sesame [Broekstra et al., 2002] architecture, and present the results of several benchmark tests we have undertaken with our approach. Additionally, we introduce an alternative algorithm for truth maintenance that eliminates some of the bottlenecks found in the depedency-based TMS algorithm.

Parts of the work presented in this chapter have been published earlier, in [Broekstra and Kampman., 2003].

7.1 Introduction

RDF and RDF Schema allow the assertion of individual statements, which form a very simple fact-base that is completely monotonic with respect to addition. However, there are a number of primitives in the universe of discourse, the semantics of which lead to inferencing rules: based on the existence of certain facts, other facts can be derived. While this process is monotonic, it requires additional bookkeeping when statements are deleted: after all, the deleted statement may have been a justifying statement for one or more derived statements. Dealing with this problem is referred to as disbelief propagation [Martins, 1990], and it is typically the task of a Truth Maintenance System (TMS) [Doyle, 1979] to restore consistency and accuracy of the knowledge base when these situations occur.

In this chapter, we study a number of practical issues that arise when computing the deductive closure of an RDF Schema and dealing with the consequences of delete operations. We present a Truth Maintenance algorithm that makes use of dependencies between statements to deal with 'non-monotonous' updates (i.e. delete operations) to an RDF Schema knowledge base. We present how this algorithm has been implemented

in the Sesame [Broekstra et al., 2002] architecture, and present the results of several benchmark tests we have undertaken with our approach.

The purpose of dependency tracking between statements as presented in this chapter is twofold. First, the information about the dependency relations between statements is necessary for several high-level services on RDF repositories, such as change tracking and statement-level security (see [Kiryakov et al., 2002] for details). Second, the goal is to achieve a performance improvement in removal operations when compared to brute-force approaches.

This chapter is organized as follows: in section 7.3, we briefly introduce some related approaches and discuss their relevance in our case. In section 7.4, we introduce the Truth Maintenance algorithm. In section 7.4.5, we look at some implementational aspects. In section 7.5, we present the results of our several benchmark tests that we conducted on the Sesame implementation of the algorithm. Section 7.6 sketches an alternative algorithm that eliminates some of the bottlenecks of the dependency-based TMS algorithm, and finally in section 7.7 we present our conclusions.

7.2 Why Truth Maintenance is Necessary

The RDF model is a very simple model that deals strictly with globally true assertions, has no negation, and no closed world assumption. Therefore, RDF models are monotonous in nature. This would seem to suggest that truth maintenance is unnecessary.

However, since the RDF model deals with inferred knowledge through inference rules, care needs to be taken that removal operations are handled properly. For example, suppose that we have an RDF model M and a statement $s \in M$. Furthermore suppose that $s' \in M$ is a statement that is derived (using RDF semantics) from s. Now, when an operation removes s from M, we need to verify that s' is still derived, and if not, s' should also be removed from M.

In a situation where RDF entailment is supported in a backward chaining fashion, such truth maintenance is not necessary, since in such situations only explicitly asserted statements are stored and derived statements are only materialized at query time. However, as we have seen in chapter 6, most RDF entailment algorithms use some form of forward chaining where (part of) the deductive closure of the RDF graph is stored. In these cases, a method for determining validity of derived statements after removal operations is necessary.

7.3 Related Work

In [Doyle, 1979], Doyle proposes a Truth Maintenance System, to allow reasoning programs to make assumptions and subsequently revise their beliefs when discoveries contradict these assumptions.

The proposed system operates by recording and maintaining the *reasons* for which the program holds a belief. These reasons (or *justifications*) consist of ordered pairs of

sets of other beliefs, such that a belief is justified if each belief in the first set is justified, and each belief in the second set is not. For example, a proposition Q might have the justification $(\{P, P \rightarrow Q\}, \{\})$.

In [de Kleer, 1986], Johan de Kleer proposes an improved TMS, which is not just based on justifications, but in addition manipulates sets of assumptions. This addition makes the algorithm more efficient as many backtracking steps are avoided and the system can deal with inconsistent information more efficiently.

The system proposed in this paper is much simpler than the systems by Doyle and de Kleer: it only has to deal with propagation of disbelief (i.e. the TMS is only invoked when propositions are retracted). In fact, the algorithm that we propose is a drastically simplified and altered version of the justification-based TMS proposed in [Doyle, 1979].

Nevertheless, the truth maintenance algorithm we propose is novel in the sense that no such system (to our knowledge) has been implemented for RDF and RDF Schema sofar, and we take several practical and implementational aspects into account, which results in a solution that is not just logically sound but also practical and extendable.

7.4 Truth Maintenance

As we have seen in section 7.3, truth maintenance in RDF models is a comparitively simple problem. Since RDF is monotonic in nature, truth maintenance only becomes important when a statement is retracted. In other words, we are dealing with a single aspect of truth maintenance: *disbelief propagation*[Martins, 1990].

Our approach is further simplified by a practical assumption that is being made: only explicit statements can be retracted. Retracting implied statements would amount to retracting at least one premise statement of every proof of the statement (if not, retraction would be pointless since the statement would just be inferred again). Since there may be many proofs, each of which have many premises, the action is no longer uniquely defined – which premise statement of a proof should be retracted? Consequently, user interaction might be required to determine the action of the system.

7.4.1 A brute-force approach

In our setting, where the complete closure is stored by means of a forward chaining inferencer, truth maintenance involves 'physical' retraction of statements that are no longer justified. A brute-force algorithm involves, quite simply, discarding all statements that were inferred and re-computing the closure.

The obvious benefit of the brute-force approach is that no additional bookkeeping is necessary, apart from whether a statement is explicit or derived.

A downside to this approach is that it makes even simple delete operations computationally expensive (see table 7.5 for comparitive results).

7.4.2 A justification-based TM algorithm for RDF

In this section, we present a justification-based Truth Maintenance algorithm. The algorithm makes use of the dependency relations between entailment rules (see table 6.3). More specifically, it tracks, for each statement in the model, of which other statements it is dependent, or put another way, which other statements *justify* it (cf. [Doyle, 1979]).

As we have mentioned earlier, the purpose of dependency tracking is twofold: the metadata thus acquired is necessary for services such as repository change tracking and statement-level security policies (as described in [Kiryakov et al., 2002]), and furthermore the aim is to achieve an improvement in the performance of removal operations when compared to brute-force approaches.

The algorithm will be executed at the end of any update transaction $\mathcal{T}$ which contains one or more delete operations.

We call B the set of believed facts. A fact $f \in B$ is labeled $explicit$ if it is asserted explicitly. If it is not explicitly asserted, f is labeled $derived$.

S is the set of justifications. Each justification $s \in S$ is of the form $\langle f_s, d1_s, d2_s \rangle$: $f_s, d1_s, d2_s \in B$. f_s is the fact justified by s. $d1_s$ and $d2_s$ are justifying facts for f_s, where $d1_s$ corresponds to the first premise of the entailment rule that produced s, and $d2_s$ to the second premise (note that the entailment rules always have at most two premises). When f_s is an RDF MT axiom statement, $d1_s = \alpha$. When s is justification for an explicit fact $d1_s = \emptyset$. In both cases, and when s was produced by a rule with a single clause in the premise, $d2_s = \emptyset$.

Furthermore, we introduce a set D wich contains *suspended* statements.

Definition 7 *A* suspended *statement is a statement that is a candidate for removal from the repository during a transaction.*

Initially, this set contains all statements f on which delete operations were performed in $\mathcal{T}$.

The initial truth maintenance algorithm is shown in table 7.1. The first loop (line 1-8) in the algorithm is an initialization step that labels each explicit statement that was removed in $\mathcal{T}$ as $derived$, and removes the corresponding justification from S. Notice that if a suspended statement is not explicit, it is removed from D, because derived statements can not be deleted except by deleting the explicit statements that justify them.

After this initialization, the algorithm enters a loop where it scans all suspended statements f (line 11). It determines the justification for each f (line 12), and if there is no justification then first the statement is removed from both B and D, and second each justification which contains the statement is removed as well (line 16-18). Since we are removing justifications, the derived statements that were justified by these justifications need to be re-examined, so they are added to the set of suspended statements (line 19-21). The algorithm terminates when there are no more suspended statements, or when a complete pass is made without removing any statements from B.

```
01.  for each f ∈ D:
02.   if (f = explicit) then
03.      label f derived;
04.      remove s⟨f, ∅, ∅⟩ from S
05.   else
06.      remove f from D;
07.   endif;
08.  end for;
09.  repeat
10.   let removed := false;
11.   for each f ∈ D:
12.      if (∀s⟨f_s, d1_s, d2_s⟩ ∈ S : f_s ≠ f) then
13.         remove f from B;
14.         remove f from D;
15.         let removed := true;

16.         for each q⟨f_q, d1_q, d2_q⟩ ∈ S:
17.            if d1_q = f or d2_q = f then
18.               remove q from S;
19.               if (f_q = derived) then
20.                  add f_q to D;
21.               end if;
22.            end if;
23.         end for;
24.      end if;
25.   end for;
26.  until (D = ∅) or (removed = false).
```

Table 7.1: Initial truth maintenance algorithm

7.4.3 Cyclic dependencies and grounded justifications

Unfortunately, the initial algorithm shown in table 7.1 fails to take the occurrence of cyclic dependencies in S into account. Consider, for example, the following set of RDF statements:

```
1. (rdfs:subClassOf, rdfs:domain, rdfs:Class) (derived, axiom)
2. (my:foo, rdfs:subClassOf, rdfs:Resource)   (explicit)
3. (my:foo, rdf:type, rdfs:Class)             (derived)
```

S now contains a.o. the following elements: $a\langle 1, \alpha, \emptyset\rangle$ (produced by RDF MT axiom assertion), $b\langle 2, \emptyset, \emptyset\rangle$ (produced by explicit assertion of statement 2), $c\langle 2, 3, \emptyset\rangle$ (produced by RDF MT rule 7a) and $d\langle 3, 1, 2\rangle$ (produced by RDF MT rule 2).

If now we execute an update transaction where statement 2 is removed, the TM algorithm first marks 2 as $derived$ and removes b from S. It then checks whether S contains a justification for statement 2. In the above, it finds it: c. The algorithm then concludes that statement 2 is still justified and does not remove it. This, however, is incorrect because c contains statement 3 which is itself dependent on statement 2 again (justification d). It is clear that 2 should have been removed and that justification c should not have been taken as sufficient evidence.

The problem lies in the fact that one of the justifying statements of c is itself dependent on the justified statement. Therefore, we introduce the concept of a *grounded* justification.

Definition 8 *A justification $s\langle f_s, d1_s, d2_s\rangle \in S$ is called* grounded *if and only if neither $d1_s$ nor $d2_s$ are justified solely by f_s or by any other statement transitively justified solely by f_s.*

The rationale behind this definition is simple: if a fact is explicit, it holds no matter what the other justifications for it are. If it is derived, however, the justification for it should not be completely dependent on itself.

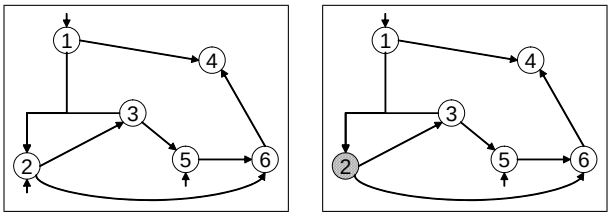
We enhance the original algorithm to capture the problem of cyclic dependencies by, after labeling deleted statements as $derived$ and removing their corresponding justifications from S, computing a new set $G \subseteq S$ which contains the *grounded* justifications g. G is computed using a mark-and-sweep-like algorithm shown in table 7.2.

We begin with a set that contains only justifications for explicit facts (since these are always justified) and axiom facts (since these are never dependent on other statements) (line 1-5). Notice that we assume that the justifications produced by explicit assertion for statements that are to be deleted are no longer present in S. From this basis, we iteratively add those justifications from S from which the justifying facts are already justified by G (line 6-14). This way, it is ensured that no justifications are added which are dependent on statements which themselves are no longer justified.

We will illustrate how this algorithm works by an example. In figure 7.1 a, the set S is depicted as a graph. Each node number refers to a statement identifier, and each arc is a justification, for example, the justification $\langle 3, 2, \emptyset\rangle$ is depicted as a directed arc from node 2 to node 3. As we can see, there are six statements. Statement 1, 2 and 5 are

```
01.  for each s⟨f_s, d1_s, d2_s⟩ ∈ S:
02.   if (d1_s = α or d1_s = ∅) then
03.      add s to G;
04.   end if;
05.  end for;
06.  repeat
07.   let newfound := false;
08.   for each s⟨f_s, d1_s, d2_s⟩ ∈ S:
09.      if [(∃t⟨f_t, d1_t, d2_t⟩ ∈ G : f_t = d1_s)
                ∧ ((∃u⟨f_u, d1_u, d2_u⟩ ∈ G : f_u = d2_s) ∨ d2_s = ∅)] then
10.         add s to G;
11.         let newfound := true;
12.      end if;
13.   end for;
14.  until newfound = false;
```

Table 7.2: Computing the grounded justifications



(a) initial situation (b) removing statement 2

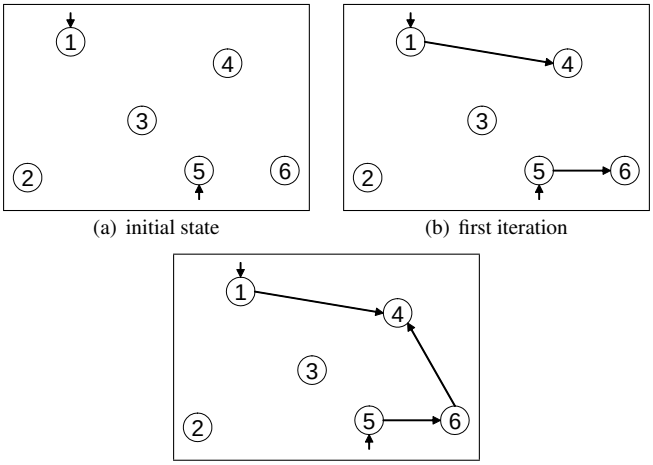
Figure 7.1: The set S as a graph

justified "out of the blue" (that is, they are either axioms or explicit statements), the other statements are justified by one or two other statements. Notice that statement 2 has an incoming arc that originates from two other nodes (1 and 3). This ternary relation depicts the justification $\langle 2, 1, 3\rangle$.

Now suppose we wish to perform an update which consists of the deleting of statement 2.

In figure 7.1 b, we see the first step: the justification for statement 2 as an explicit statement $\langle 2, \emptyset, \emptyset\rangle$ is removed. Having done this, we need to calculate the set G, which we do by adding edges to a graph which is empty at the start.

Consider figure 7.2 a. The set G is empty, except for the justifications for explicit and axiom statements. This corresponds to lines 1-5 in the mark-and-sweep algorithm above. Then, we loop over each justification in S and see if can add it to G. In the first iteration (figure 7.2 b), we come across the justification $\langle 4, 1, \emptyset\rangle$. The algorithm checks



(a) initial state (b) first iteration

(c) final iteration

Figure 7.2: Building up the grounded justification set G

if the dependency 1 is justified by G (line 9), which it is (this corresponds to node 1 having an incoming arc in G), so this arc is added to G. The same goes for $\langle 6, 5, \emptyset \rangle$. The algorithm also examines the arcs leading to nodes 2 ($\langle 2, 1, 3 \rangle$) and 3 ($\langle 3, 2, \emptyset \rangle$), but both are rejected from G, because neither node 2 nor node 3 have any incoming arcs in G.

In the next iteration (figure 7.2 c) the set G is completed by adding $\langle 4, 6, \emptyset \rangle$. In the end, no justifying arcs for node 2 and 3 were added at all, so these two statements will be removed completely by the TMS algorithm. Also notice that although nodes 5 and 6 have both lost a justification (because these justifications were dependent on nodes 2 and 3), they still are justified by other arcs and therefore will not be removed by the TMS algorithm.

Because we now have the set G, which is equal to the set of justifications that hold after completion of $\mathcal{T}$, we can simplify the rest of the TM algorithm. We introduce a new set $E \subset S$ which is the set of all expired justifications: $E = S - G$.

The enhanced TM algorithm is shown in table 7.3.

The algorithm in table 7.3 is a modification of the original algorithm: after labeling each deleted statement $derived$ and removing the corresponding justifications from S (line 2-8), the sets G (see table 7.2) and E are computed (line 9-10). Since we now have complete knowledge on justification for all statements in B, we can eliminate the `repeat..until` loop from the original algorithm completely: all statements justified by expired dependencies are added to D (line 12) and all expired dependencies are removed from S (line 13). Then, all statements in D are verified, that is, they are deleted if and only if there is no justification for them in G (line 15-19). After looping over D once, the algorithm terminates.

```
01.  for each f ∈ D:
02.   if (f = explicit) then
03.      label f derived;
04.      remove s⟨f, ∅, ∅⟩ from S
05.   else
06.      remove f from D;
07.   endif;
08.  end for;
09.  determine G;
10.  let E := S − G;
11.  for each s_e⟨f_e, d1, d2⟩ ∈ E :
12.   add f_e to D;
13.   remove s_e from S;
14.  endif;
15.  for each f ∈ D:
16.   if (∀s⟨f_s, d1_s, d2_s⟩ ∈ G : f_s ≠ f) then
17.      remove f from B;
18.   end if;
19.  end for.
```

Table 7.3: Mark-and-Sweep Truth Maintenance algorithm

7.4.4 Complexity

The algorithm presented in figure 7.3 has a complexity in the order of the number of deductive dependencies, $O(|S|)$. Since the size of S is determined by the number of entailments per statement, this number can become very high. This is a clear bottleneck and makes the algorithm less suited for the purpose of optimizing speed of delete operations: an alternative algorithm might avoid this bookkeeping task of computing and maintaining the deductive dependencies altogether.

However, since our other prime motivation for the algorithm is having the set of deductive dependencies available for higher-level reasoning tasks such as repository change tracking and statement-level security policies, there is a tradeoff to consider. Our current practical approach has been to accept the performance bottleneck in the computation of the set of dependencies in favor of enabling higher-level reasoning.

7.4.5 Implementation issues

In this section we will discuss several practical issues that arose when implementing the TM algorithm in the Sesame [1] [Broekstra et al., 2002] system.

Justification Inferencing

The algorithm tacitly assumes that the set of justifications, S, is known. However, like the closure of the set of statements, determining S requires inferencing over the set of RDF MT entailment rules from [Hayes, 2004]. Unlike for the statement inferencing, however, optimizations that skip redundant inferences are not possible in this case, since it is important that S is complete.

In the test setup in Sesame, the determination of S is therefore implemented as a separate inferencing task. The justification inferencer is invoked after the complete closure has been computed and stored. It employs a basic backward chaining strategy: it loops over all statements and determines for each one whether it is a possible conclusion of an RDF MT entailment rule, and if so, which other statements satisfy the premise(s) for the matching rule. While this computation is expensive, it is only necessary to perform it once, at some time after new statements are added to the repository.

In the test setup, the justification inferencing is done directly after statements have been added, as part of the transaction. This means that the transaction takes longer. An alternative approach would be to delay the justification inferencing, until the moment the TMS (or e.g. the versioning functionality) needs the information, or to do this part of the inferencing in a background process, locking delete operations until it is complete but leaving other types of operations on the repository available.

[1] See `http://www.openrdf.org/`

data set	set size	add statement (norm)	total upload (norm)
OpenWine	5289	1.27	1.34
SUO	12498	10.50	10.25
CIA	30260	1.07	1.11
Wordnet	373485	1.36	1.36

Table 7.4: Performance of TMS-based statement adding, normalized against the brute-force approach

data set	set size	remove (avg.) (norm)
OpenWine	5289	0.73
SUO	12498	1.08
CIA	30260	0.15
Wordnet	176037	1.68

Table 7.5: Performance of TMS-based statement removal, normalized against the brute-force approach

7.5 Results

We have run a number of tests on general performance with both the brute-force approach and the justification-based approach, with the data sets introduced in chapter 6, section 6.4.3.

The tests were carried out using Sesame release 0.8, with the SQL92Sail and the TmsSQL92Sail for brute-force and justification-based inferencing respectively. In reporting the results we normalized the figures. This is done because our chief interest is comparing the two approaches rather than doing absolute performance tests, and also because the performance figures are highly dependent the version of Sesame, the used hardware, and the DBMS configuration.

In table 7.4 the comparative performance results for uploading data sets are shown, normalized against the brute-force approach. Uploading consists of inferring and storing statements, and in the case of the TMS-enabled setup, it also includes computing the set of deductive dependencies.

As was to be expected, overall performance on upload is slightly worse in the justification-based approach, which is mainly caused by the computation of the set of deductive dependencies. We notice that the SUO data set behaves differently from other sets due to its composition: it consists exclusively of a large class hierarchy that is both broad and deep, leading to an exceptionally high size increase when computing the closure.

In table 7.5 the performance results for removing data are shown. The average given is the average per removal operation (note that this does not necessarily correspond to one removed statement, one removal operation may delete several statements).

From these results we can observe that justification-based removal performs signifi-

cantly better than its brute-force counterpart, except on the SUO data set and the Wordnet data set, in which cases it performs worse.

The reason the Wordnet and SUO sets perform worse when using the justification-based approach can be explained by the physics of the algorithm implementation and the testing environment. In the Sesame implementation of the TM algorithm, the set of grounded justifications G is computed using a rather complex SQL query that performs multiple joins with the table that stores the set of justifications, S. In the cases of SUO and Wordnet, this set of justifications is comparatively large – in the SUO test because there are so many new statements inferred (see table 6.4), in the Wordnet test because the data set itself is fairly large.

It turns out that the MySQL RDBMS used in the tests has limited capabilities for efficiently processing the SQL queries used to determine G when such large tables are involved, and its performance on such queries drops dramatically when the size of the table that stores D exceeds a certain threshold. The brute-force approach, which does not need to compute G, only has to use the SQL queries needed to do inferencing. However, these SQL queries are relatively simple and MySQL has less problems with evaluating these efficiently, even on large data sets. The upshot is that in these cases, re-computing the entire closure becomes more efficient than using the TMS approach.

Several strategies can be used to improve performance of the TMS, such as table optimization, indexing, or even the use of a more sophisticated query planner than the one available in MySQL. However, a more structured solution lies in the adjustment of the truth maintenance algorithm itself. In recent testing results, it has turned out that the strategy for computing the set G can be adapted to perform significantly better in situations where the number of removed statements in a transaction is lower than 10% of the total size of the data set. In future work we will further investigate and implement this strategy, and the implementation will be able to switch between strategies depending on the number of removed statements in a transaction.

7.6 An Alternative Approach

The J-TMS algorithm presented in the previous section uses the explicitly present dependencies. However, as we have seen, a major bottleneck of the approach is the offline computation of these dependencies. We have sketched scenarios where these dependencies are necessary for other tasks, however, other, perhaps more common, scenarios do not require this information to be present.

An alternative truth maintenance algorithm therefore does not rely on precomputed dependency inf:ormation, but rather uses online backward chaining to determine statement dependency dynamically, for each removed statement, during the transaction in which the removal operation occurs. In this section, we present an outline of this algorithm.

The algorithm will be executed at the end of any update transaction $\mathcal{T}$ which contains one or more delete operations.

Let B be the set of believed facts. A fact $f \in B$ is labeled $explicit$ if it is asserted explicitly. If it is not explicitly asserted, f is labeled $derived$. Furthermore we call set D

the set of *suspended* statements (see definition 7). We also define an implication operator $\vdash_n$:

Definition 9 *$f \vdash_n f'$ iff f' can be derived from f through n applications of RDF entailment rules.*

```
01.  add each f ∈ B which is deleted in T to D
02.  mark all f ∈ D derived in B
03.  while D ≠ ∅:
04.   for each f ∈ D:
05.      let V = ∅
06.      if not impliedByProofSystem(f, V) then
07.         remove f from D
08.         for each f' ∈ D : f ⊢1 f':
09.            add f' to D
10.         endfor
11.      endif
12.      remove f from D
13.   endfor
14.  endwhile
```

Table 7.6: Main outline of a backward chaining TMS algorithm

The global outline of the backward chaining TMS algorithm is shown in table 7.6. As can be seen, the algorithm loops over the set of suspended statements until it is empty, verifying for each suspended statement f whether it is still implied by the proof system, and if not, removing it from the model and suspending each statement f' that can be derived from f in a single step.

7.6.1 Goal Driven Entailment over RDF Semantics

The RDF semantics define a set of 13 entailment rules (see chapter 6, section 6.2). These rules are defined in a data-driven manner. Goal-driven reasoning simply 'reverses' these rules: the algorithm examines each statement goal for a match with a rule consequence and tries to determine whether matching premises for a matching rule can be found.

Using these reformulated rules we can specify a recursive algorithm for goal-driven RDF reasoning. Let V be a set of facts that are visited during the run of the algorithm. The algorithm is shown in table 7.7.

While recursive application of the entailment rules is, at first sight, an expensive and not very scalable solution, it is important to realize that in this particular algorithm, the recursive step takes place over the backtrace of a deduction path for a *single* statement (in comparison, the forward chaining reasoner presented in chapter 6 iterates over large sets of statements). The recursive descent terminates as soon as a proof has been found,

```
impliedByProofSystem(f, V):
      if f ∈ V then // loop detection
         return false
      endif
      if f = explicit then // stop condition
         return true
      endif
      add f to V
      if f.predicate = type ∧ f.object = Property then // matches rule 1
         if (∃f' ∈ B : f'.predicate = f.subject) then
            if impliedByProofSystem(f', V) then
               return true
            endif
         endif
      endif
      if f.predicate = type then
         if (∃f' ∈ B : f'.predicate = domain
            ∧f'.object = f.object) then // matches rule 2 first premise
            if (∃f'' ∈ B : f''.predicate = f'.subject
               ∧f''.subject = f.subject) then
               if (impliedByProofSystem(f', V) ∧
                  impliedByProofSystem(f'', V)) then
                  return true
               endif
            endif
         endif
      endif
      //(etc.  for all entailment rules)
      return false // since no deduction was found
```

Table 7.7: Recursive goal-driven inferencer for RDF Semantics

or all possible back traces have failed. Again, we see that the search space is pruned by examining the subject, predicate and/or object of the statement under investigation.

The set V is introduced to prevent the algorithm from entering an endless loop. In section 7.4.3, we have seen that dependencies between statements in an RDF graph can be cyclic. By keeping a list of statements that were visited during the recursive descent, we break such loops. Since the recursion only applies to a single statement, this list will not become very large.

An interesting observation that we have made in our experiments with the iterative forward chaining algorithm presented in chapter 6, is that the number of iterations it takes to complete the closure of an RDF model is typically quite low, and can be estimated by

the depth of the class hierarchy.

Theorem 4 (Iterations Worst Case) *For an RDF model with a class hierarchy of depth n and a property hierarchy of depth m, the number of iterations over the entailment rules to compute the full closure is at most $max(n, m)$.*

We will show that theorem 4 holds by means of an example. In figure 7.3, we see a simple class hierarchy of depth 3. The dotted arrows are entailed subClassOf relations. In the first iteration, it is established (through rule 11) that D is a subclassof A ($D \subset A$), and $F \subset B$. In the second iteration, we determine $F \subset A$ as well (since now we know $F \subset B$, and $B \subset A$).

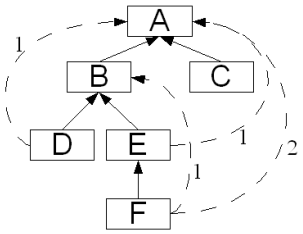


Figure 7.3: Closure of a hierarchy of depth 3 in 2 iterations

Propagation of type relations works in a similar fashion: assume a is an instance of F: $a \in F$. In the first iteration, through application of rule 9, we establish $a \in E$. In the second iteration, we establish $a \in B$, and also $a \in A$ (since we know that $a \in E$ and $E \subset A$). Thus, closure of class hierarchy and type propagation is established in $n - 1$ iterations.

After the class hierarchy and type relations have been entailed, there may be additional entailment to be made that are not related to class and property subsumption. However, if we examine the entailment rules, we can establish that in the worst case this will take no more than one iteration: if we eliminate the rules that deal with class and property subsumption, we only have rules 1, 2, 3, 4a and 4b left. Of rules 1 and 4 we have already established that they are only entailing new triples in the first iteration (see chapter 6, section 6.4.1), this only leaves rules 2 and 3 to consider. These rules derive type relations based on domain/range constraints. It is trivial to see that neither rule provides a useful input for the other, since they only derive type relations, and both the domain and range of `rdf:type` are axiomatically known (`rdfs:Resource` and `rdfs:Class`, respectively). Therefore, determining the closure of a model with no class/property hierarchy left to consider never takes more than 1 extra iteration.

We have now established that the worst-case depth of the recursion of the algorithm in table 7.7 is equal to $max(n, m)$ where n and m are the depths of the class and property hierarchies, respectively.

The streaming, per-statement nature of the alternative TMS algorithm and the relative short paths of entailment in RDF models (and thus relatively shallow recursion)

make it an attractive algorithm for supporting truth maintenance when explicit presence of all dependencies between statements is not necessary. The elimination of offline dependency computing makes the algorithm a lot more scalable and performant. Actual implementation and experimental testing of this algorithm remain future work, however.

7.7 Conclusions

In this chapter we have addressed several issues concerning RDF truth maintenance. We have argued that removal operations are important to consider in relation to the inference strategy employed, and we have presented an algorithm for capturing dependencies between statements and exploiting these dependencies for doing truth maintenance.

We have shown that the truth maintenance algorithm using dependencies between statements performs quite well on medium-sized data sets, and an improvement of removal operation performance as compared to brute-force approaches is achieved on these data sets.

However, though the Sesame system itself can easily cope with data sets that consist of over 30 million statements, it seems apparent from the data obtained from in particular the Wordnet test that remove operations as implemented in the test system will not scale well to these high figures. A proviso here is that the test results were obtained using unoptimized code in the Sesame system, recent tests with more optimized code have actually significantly improved absolute performance, in many cases by a factor 10 or more. These results, however, are not presented here because they cannot be compared to the brute-force approach, for the following reason: Sesame now makes use of the TMS algorithm internally, and there is no brute-force counterpart implemented to obtain a valid comparison.

Regardless, we conclude that although this TMS approach performs satisfactorily for medium-sized data sets and the current expressiveness of RDF and RDF Schema, the approach will have to be adjusted to cope with larger data sets. In particular, the justification-based truth maintenance algorithm can be further tuned to perform better under difficult circumstances by better indexing schemes and possibly using a more sophisticated query planner, as well as by adapting the computation of G, and switching strategies depending on the statistics of the transaction (in fact some of these optimization have already been implemented, resulting in the aforementioned performance increase).

An alternative implementation of inference, such as the schema inferencing algorithm presented in chapter 6, will make use of more sophisticated closure computing algorithms and will forgo keeping track of all deductive dependencies. This will result in the loss of capabilities for change tracking and security policies but will result in faster performance and a smaller memory footprint.

Since not every task will require the higher-level services mentioned earlier, an alternative truth maintenance algorithm has been introduced in section 7.6. We have shown how this algorithm has attractive complexity properties and eliminates the overhead of dependency tracking by using dynamic goal-driven inference to determine dependencies between statements. Future work remains the implementation and experimental testing of this alternative algorithm.

Chapter 8

A Case Study in Distributed Querying and Data Integration

In the previous chapters, we have introduced frameworks and storage systems primarily designed for centralized processing of RDF. In this chapter, we will investigate how the presented languages, tools and strategies can be applied in the real world. We will present general approaches for dealing with distributed querying and data integration and illustrate how ontologies can play a supporting role. We then look at the DOPE project, a case in which the Sesame framework has been used in order to realize an ontology-based document retrieval system for information analysts in the pharmaceutical industry.

Parts of the work presented in this chapter were published earlier in [Stuckenschmidt et al., 2004a] and [Stuckenschmidt et al., 2004b].

8.1 Introduction

The notion of distributed querying is well established in databases (cf. [Rothnie et al., 1980]). In general, distributed querying deals with evaluating a single query over multiple data sources.

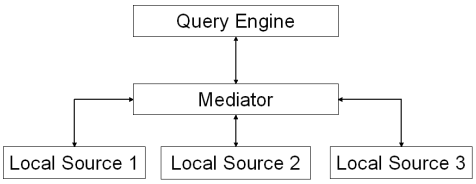


Figure 8.1: Generic Distributed Querying Architecture

In figure 8.1, we see a generalized representation of a distributed querying setup. Each local source is known by a *mediator*, which distributes the incoming query over the

local sources, collects partial results and sends the result back to the query engine. The mediator acts as an abstraction layer such that, from the point of view of the querying client, the collection of queried local sources acts as a single database.

Distributed querying can be used in several scenarios, two of which we briefly describe here:

- **load balancing**
  On a heavily queried database system, several duplications of the local source on different machines can help distribute the load of evaluating queries. In such scenarios, the mediator plays the role of *least load scheduler*, routing each incoming query to a local source that is currently idle. Each local source is an exact copy of the other.

- **integration of data sources**
  Different data sources may exist at different physical locations. To be able to query such different data sources, a single interface that unifies these sources in a 'virtual' database is necessary. The mediator now plays the role of a virtual database. Depending on the level of sophistication of the setup, the mediator can inspect incoming queries, splitting these and sending relevant subqueries to each local source, integrating the local results back into a single result that answers the original query.

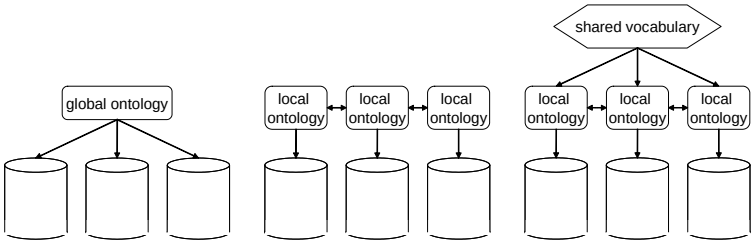
The need for handling multiple sources of knowledge and information is quite obvious in the context of Semantic Web applications. First of all we have the duality of schema and information content where multiple information sources can adhere to the same schema. Further, the re-use, extension and combination of multiple schema files is considered to be common practice on the Semantic Web (compare [Hendler, 2001]). In the following section, we briefly summarize approaches for the use of ontologies in data integration tasks.

8.2 Using Ontologies for Data Integration

The problem of integrating heterogeneous data sources has been addressed by many researchers (see [Levy, 1999] or [Wache et al., 2001] for surveys). In [Wache et al., 2001], the use of ontologies for integration of heterogeneous data sources is reviewed. The authors present the different way in which ontologies can be used in this context and survey existing systems. We summarize their findings in this section.

Generally speaking, ontologies can be used to describe the semantics of information sources. They can assist in the integration task by identifying and associating semantically corresponding concepts in different sources. We can identify three main approaches towards employing ontologies for information integration (see figure 8.2):

- The *single ontology approach* uses a global ontology providing a single vocabulary the specification of the semantics. All information sources are directly coupled to this this one global ontology. This approach can be applied to problems where the



(a) single ontology approach (b) multiple ontology approach (c) hybrid ontology approach

Figure 8.2: Three possible approaches for content explication through ontologies (from [Wache et al., 2001]).

semantics of the information sources are relatively close in terms of the granularity of the view. A disadvantage of the approach is that it is very rigid and susceptible to change; changes in the underlying information source almost inevitably lead to changes in the global ontology.

- The *multiple ontology approach* uses a separate ontology for each information source. An advantage of this approach is that no commitment to a single global ontology is necessary, and therefore integration of additional sources becomes easier. However, the lack of a common vocabulary makes comparison and mapping of the information in different sources very difficult.

- A *hybrid approach* describes the semantics of each information source in its own ontology, similar to the multiple ontology approach. However, to make the local ontologies comparable to each other they are built from a global shared vocabulary. The shared vocabulary contains the basic terms (the primitives) of a domain. These are combined in the local ontologies in order to describe more complex examples. This shared vocabulary can sometimes be an ontology itself. A large advantage is the hybrid approach is that new information sources can be added without need for modification. The use of common vocabulary makes sources comparable.

Apart from explicating content of sources, ontologies can also be used as a global query model for the distributed setting. Using the ontology as a query model has the advantage that the structure of the query model is more intuitive to the user, because it corresponds more closely to the user's own appreciation of the domain.

8.3 Using the SAIL API for Data Integration

In the previous section we have discussed the conceptual integration of information sources by using ontologies. In this section, we will focus on the technical aspects of realizing such an integrated system, using the Sesame framework.

As we have seen in chapter 5, section 5.2.1, the flexible nature of Sesame's SAIL API allows it to be used as an abstraction layer across a wide variety of storage backends. This notion can be further extended by using the SAIL to realize a distributed storage.

Mapping the generalized architecture of figure 8.1 to the Sesame case is fairly straightforward: each local source is realized by a SAIL implementation, and the mediator is a SAIL implementation as well. This allows the functional modules of the Sesame framework to treat the entire distributed repository as a single data source, and the handling of query distribution and result composition can be handled internally in the mediator SAIL. Notice that this general architectural setup is flexible enough to be used in any of the conceptual approaches described in section 8.2.

An essential part of realizing a distributed system is the encoding of knowledge in the mediator: it has to decide how to distribute (sub)queries over sources based on the information it possesses about each source. In the Sesame framework, knowledge can be encoded in the mediator about query path expressions and how they relate to local sources. Upon receiving a query, the mediator can then inspect the query's path expressions and generate subqueries for each relevant source, receiving the partial results and composing it into a single query answer.

Another essential part is providing mappings (using one of the approaches described in section 8.2) between information sources and the ontology. The SeRQL query language can be used for transformations from one model to another, providing an effictive mechanism for defining views on each data source that map the local information source to the global schema.

8.3.1 Query Triggered Distribution

In order to be able to make use of the optimization mechanisms of the database engines underlying the different repositories, we have to forward complete queries to each repository. In the case of multiple external models, we can further speed up the process by only pushing down queries to information sources we can expect to contain an answer. The ultimate goal is to push down to a repository exactly that part of a more complex query for which a repository contains an answer. This part can range from a single statement template to the complete query. We can have a situation where a subset of the query result can directly be extracted from one source, and the rest has to be extracted and combined from different sources. This situation is illustrated in the following example.

Consider the case where we want to extract information about research results. This information is scattered across a variety of data sources containing information about publications, projects, patents etc. In order to access these sources in a uniform way, we use the OntoWeb research ontology as the global querying schema.

Figure 8.3 shows parts of this ontology. Suppose we now want to ask for the titles of articles by employees of organizations that have projects in the area "RDF". The path expression of a corresponding SeRQL query would be the following:

```
{A} title {T};
    author {W} affiliation {O};
    carriesOut {P} topic {'RDF'}
```

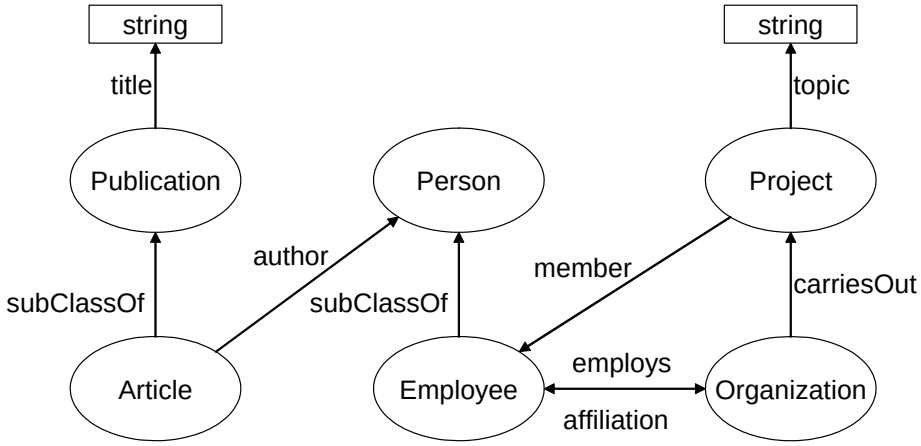


Figure 8.3: Part of the Ontoweb ontology

Now, let us assume that we have three information sources S_1, S_2 and S_3. S_1 is a publication database that contains information about articles, titles, authors and their affiliations. S_2 is a project database with information about industrial projects, topics, and organizations. Finally, S_3 is a research portal that contains all of the above information for academic research.

If we want to answer the query above completely we need all three information sources. By pushing down the complete query to S_3 we get results for academic research. In order to also retrieve the information for industrial research, we need to split up the query, push the fragment

```
{A} title {T};
   author {W} affiliation {O}
```

to S_1, the fragment

```
{O} carriesOut {P} topic {'RDF'}
```

to S_2, and join the result based on the identity of the organization.

The example illustrates the need for sophisticated indexing structures for deciding which part of a query to direct to which information source. On the one hand we need to index complex query patterns in order to be able to push down larger queries to a source; on the other hand we also need to be able to identify sub queries needed for retrieving partial results from individual sources.

8.3.2 Integrating Existing Data Sources

Using the distributed model through Sesame's SAIL API, integration of existing data sources becomes possible within the generic architecture.

The SAIL API's notion of storage backend abstraction can be used to wrap practically any data source for inclusion as a local source in the distributed setup.

For example, one could consider a distributed system where one repository contains metadata on a large number of documents. Typically, such a repository will not contain the integral text of these documents, nor will it be able to effectively index and search the text.

However, dedicated keyword indexers such as Lucene [1] excel at fast indexing of large document sets. A SAIL implementation that converts SAIL methods to Lucene actions will be able to leverage the power of Lucene's indexing and searching capabilities and use this knowledge as an integral part of the RDF query answering of our distributed system.

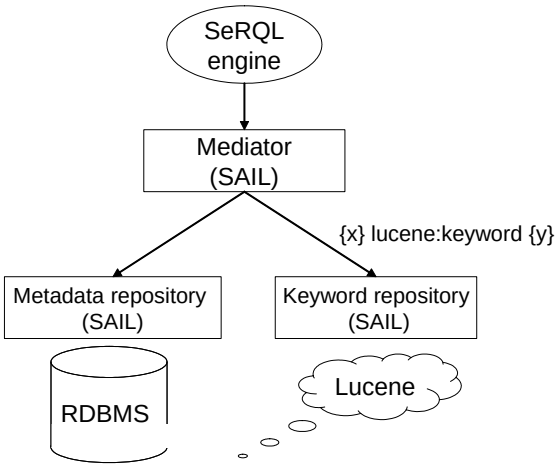


Figure 8.4: Distributed Querying of a Lucene Indexer and Metadata Repository

In figure 8.4 we see an example setup of such a distributed querying system. The system forwards all typical SeRQL queries that concern document metadata to the metadata repository, but when the mediator encounters a particular path in the query (using the `lucene:keyword` property), it additionally queries the Lucene source for keyword information to return as answer. In this particular example, we use a multiple ontology approach, where each local source has its own ontology, and the query mediator examines queries to determine which part has to be forwarded to which source.

The setup allows integration of existing sources without affecting their actual function: the Lucene indexer still functions independently from the rest of the distributed system and functions as an independent information provider. Furthermore, the setup removes the need for costly offline conversion of data from the source into RDF format, costly both in terms of required space and maintenance, since such a conversion necessarily means duplication of information.

[1] `http://jakarta.apache.org/lucene`

In the example of the Lucene indexer, integration is simplified because the Schema to which the source adheres is simple (in fact, it consists of a single property, `lucene:keyword`). In cases where the data of the external source is semantically richer, the associated schema will be larger and more complex. In these cases integration of this metadata with the schema used on the client level (i.e. the overall domain ontology of the system) may be less than trivial and a mapping from system-specific terminology to the conceptual model of the domain ontology may be required. In the next section, we will describe how the employment of SeRQL queries as view definitions can realize this.

8.3.3 View Definitions through SeRQL

In [Stuckenschmidt, 2003] it is discussed that integration of the global ontology and the model of the information source can be achieved by means of *view definitions*, which map the vocabulary of the one model to the other. Two main approaches are distinguished:

- **Global-as-View:** In this approach, every relation in the global schema is defined as a view over the different schemas that are to be integrated (see also [Garcia-Milina et al., 1997]).
- **Local-as-View:** In this approach, views are used in exactly the opposite way (see for example [Levy et al., 1996]): views define how local information maps to the global schema by expressing a mapping from each relation in the local schema to a (set of) relation(s) in the global schema.

An advantage of the global-as-view approach is that answering queries over the global schema is relatively straightforward: it requires expanding the incoming query into the terms of each local source as defined by the view definitions. However, a disadvantage is that there is a dependency between the global schema and the local sources: adding or removing sources is difficult and may result in changes to the global schema.

The local-as-view approach has as its main advantage that there is no dependency: each local source is mapped to the existing global schema, and adding new source simply requires defining the necessary mappings. However, in this approach query answering is more difficult: query expansion is no longer possible and instead an abduction-like approach is required.

The Sesame framework allows the implementation of mapping schemes through view definitions by means of the SeRQL query language. SeRQL is a closed language and as such can be used to transform graphs. Specifically, the `construct` clause allows us to:

- identify relevant subparts of the information source;
- manipulate the form of the returned result by changing relations between information items or by introducing new vocabulary and new relations.

However, this does not mean that SeRQL can be seen as a full view definition language. An important prerequisite of a view is that it is *named*: queries on the higher level need to able to refer to the defined view. No such naming mechanism exists in SeRQL.

However, if we step outside the boundaries of the query language itself and take the combination of query language and API into account, we can realize view definitions within the scope of the Sesame framework. The result of any view-defining SeRQL query can be stored in a temporary object structure (such as an in-memory Sesame repository), and this repository can then be referred to by higher-level queries.

It is important to realize that both the global-as-view and the local-as-view approach can be supported in this fashion. The former approach can be realized by mapping each relation in the global schema to a specific 'view-repository'. The local-as-view approach can be realized by having each local source be transformed to a model in terms of the global schema and stored in a single 'temporary' repository that is used for querying in terms of the global schema.

Materialization of the view is a choice that can be made with respect to the scenario. For views where large amounts of data are involved, a lazy evaluation may be necessary: the view-defining SeRQL query is evaluated at the moment that the view's data is necessary, not before. In settings where memory conservation is not an issue and the main priority is querying speed, offline materialization of the view is possible. Of course, in such settings, special care has to be taken to keep the view up to date with the original data.

In the next section, we will describe a case study where the querying of distributed data through the Sesame framework, using SeRQL queries as view definitions, has been realized.

8.4 Case Study: the DOPE project

The general distributed architecture outlined in the previous section was applied in practice in the system developed for the DOPE project [Stuckenschmidt et al., 2004a]. In this section, we describe the project and the role distributed querying plays in it.

With the unremitting growth of scientific information, integrating access to all this information remains an important problem, primarily because the information sources involved are so heterogeneous. Sources might use different syntactic standards (syntactic heterogeneity), organize information in different ways (structural heterogeneity), and even use different terminologies to refer to the same information (semantic heterogeneity). Integrated access hinges on the ability to address these different kinds of heterogeneity.

Also, mental models and keywords for accessing data generally diverge between subject areas and communities; hence, many different ontologies have emerged. An ideal architecture must therefore support the disclosure of distributed and heterogeneous data sources through different ontologies. To serve this need, the DOPE project (Drug Ontology Project for Elsevier), has explored ways to provide access to multiple information sources through a single interface. The result, the DOPE system, is a thesaurus-based search system that uses automatic indexing, RDF-based querying, and concept-based visualization.

8.4.1 Thesaurus-based Information Access

Thesauri have proven to be essential for effective information access. They are hierarchically organised controlled vocabularies that can be used for indexing information and thereby help to overcome many free-text search problems by relating and grouping relevant terms in a specific domain. Thesauri in the life sciences include MeSH, produced by the US National Library of Medicine[2] and EMTREE, Elseviers life science thesaurus[3].

These thesauri provide access to information sources (in particular document repositories) such as PubMed[4] and EMBASE.com, but no open architecture exists to support using these thesauri for querying other data sources.

Elsevier maintains the EMTREE thesaurus as a terminological resource for life science researchers. EMTREE is used to index EMBASE, a human-indexed online database. EMTREE currently contains the following information types:

- *Facets* are broad topic areas that divide the thesaurus into independent hierarchies.
- Each facet consists of a hierarchy of *preferred terms* used as index keywords to describe a resources information content. Facet names are not themselves preferred terms, and they cannot be used as index keywords. A term can occur in more than one facet; that is, EMTREE is poly-hierarchical.
- Preferred terms are enriched by a set of *synonyms* – alternative terms that can be used to refer to the corresponding preferred term. A person can use synonyms to index or query information, but they will be normalized to the preferred term internally.
- *Links*, a subclass of the preferred terms, serve as subheadings for other index keywords. They denote a context or aspect for the main term to which they are linked. Two kinds of link terms, drug-links and disease-links, can be used as subheadings for a term denoting a drug or a disease.

EMTREE 2003 contains about 45,000 preferred terms and 190,000 synonyms organized in a multilevel hierarchy. The EMTREE thesaurus serves primarily as a normalized vocabulary for matching user requests against documents in the target sources. This project uses natural language technology provided by Collexis[5] to automatically index documents in several different repositories with keywords from EMTREE. A Collexis *fingerprint* server houses the results and can be queried via a SOAP interface. (A Collexis fingerprint is very small representation of the characteristic concepts in a piece of source text.)

Natural language frequently refers to the same concept in several ways. The SOAP interface contains an indexing engine that uses EMTREEs synonym relations to return keywords most likely to be relevant to a given search input string. Also, EMTREEs hierarchical relations can identify keywords more specific than the target keyword, letting

[2] `http://www.nlm.nih.gov/mesh/meshhome.html`
[3] `http://www.elsevier.com/homepage/sah/spd/site`
[4] http://pubmed.org/
[5] `http://www.collexis.com/`

users expand their searches and thus gain much better recall. The results are ordered by relevance.

Among our challenges was identifying the minimal set of metadata (from each source) to be stored. The user interface assumes that several metadata are available for retrieval or display. The DOPE prototype uses indexes of the full content of ScienceDirect (full-text articles) and the last 10 years of Medline. These sources have different sets of metadata, and future DOPE versions will standardize them using the Dublin Core Metadata Initiative[6]. In general, however, DOPE permits easy inclusion of new data sources.

8.4.2 RDF-based Information Access

To provide the integration functionality introduced in the previous section, we need a technical infrastructure to mediate between the information sources, thesaurus representation, and document metadata stored on the Collexis fingerprint server. We implemented this mediation in the DOPE prototype using the Sesame framework.

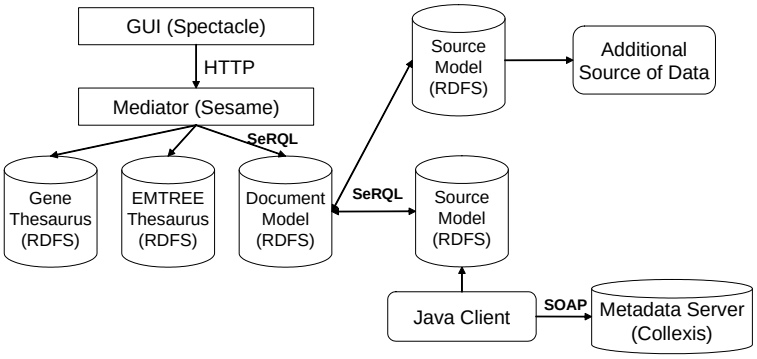


Figure 8.5: DOPE's Conceptual architecture

Figure 8.5 shows the DOPE architecture. It consists again of a central mediator component and a number of local information sources, all in the form of Sesame repositories (and thus using Sesame's SAIL API for access). Elseviers main life science thesaurus, EMTREE 2003, was converted to an RDF schema format and added to a Sesame repository directly. Using EMTREE 2003 and the Collexis fingerprinting technology, several large data collections were indexed (five million abstracts from the Medline database and about 500,000 full-text articles from Elseviers ScienceDirect). This information was stored in the external metadata server and available through a SOAP interface.

DOPE dynamically maps the Collexis metadata to an RDF model in two steps. The first step creates an RDF model, an exact copy of the data structure provided by the fingerprint server, and stores it in the source model repository. In the second step, this

[6] http://dublincore.org

intermediate RDF model is mapped to the final conceptual model used for querying the system, and stored in the document model repository. In the next section, we will explain the dynamics of this system in more detail.

8.4.3 Model Transformation

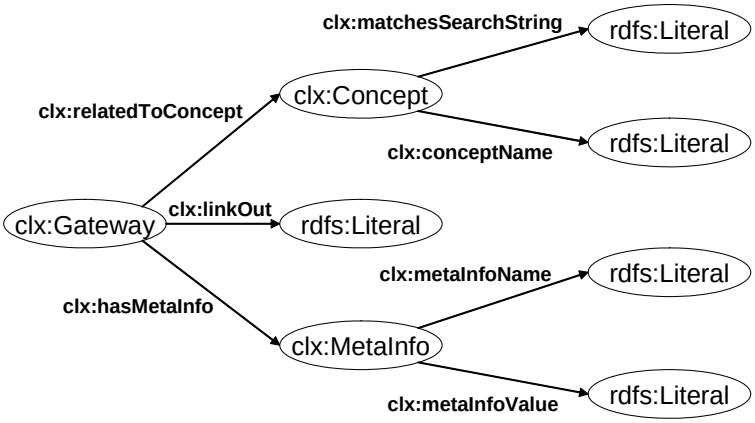


Figure 8.6: The physical data model, using Collexis terminology

The external Collexis server is not equipped with RDF-based input and output facilities. The DOPE prototype therefore deploys an extractor component that uses the Collexis SOAP interface to convert the available information to RDF, creating a physical model (Figure 8.6) that is a 1:1 mapping to the original information.

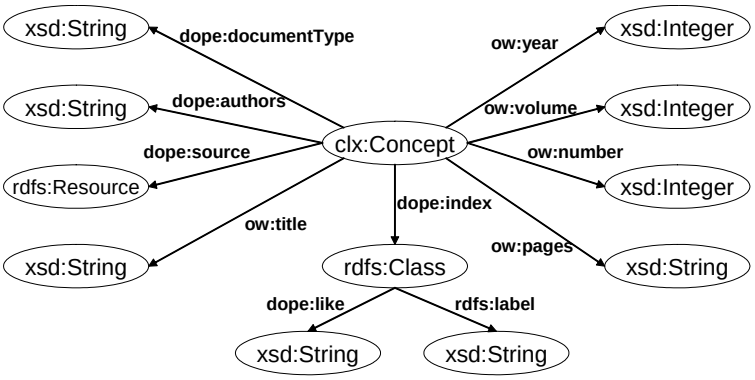


Figure 8.7: The logical data model, using OntoWeb Document Ontology terms

Although the physical model is already in RDF, it is not in the terminology in which

the queries are formulated. Moreover it is not well suited to direct merging with different data sources, since the terminology is very specific to the Collexis metadata extraction process. To transform the RDF data to a different, more conceptually oriented model, the SeRQL query and transformation language was used. SeRQL queries are used transform the physical model into a logical model (see Figure 8.7). The logical model is based on an adapted subset of the OntoWeb ontology[7] and is particularly well-suited to a conceptual representation of documents that makes no assumptions about the source of the data. The Ontoweb ontology was adapted by linking the model to the schema used to represent the EMTREE thesaurus. This link appears in the lower part of Figure 8.7: each publication links to an RDF schema class that represents a preferred term in the thesaurus. Each publication is also annotated with a label and a relation to similar search strings that the Collexis server computes on the fly when it processes a query.

8.4.4 Triggered Prefetching

In the previous section the process of transforming the data available in the external metadata server to RDF was introduced. This is a dynamic process that happes at runtime, during query evaluation, and it allows the system to only convert those parts of the external metadata that are relevant to the current query. Although this may at time result in slow query processing, the alternative, offline static conversion of the entire dataset, has a number of distinct disadvantages: first of all, offline conversion implies duplication (and storage) of data, which in the case of the metadata server would mean a huge additional repository. Second, in scenarios where updating of information is an issue, having a duplicate around severely complicates matters.

The system uses a notion of *triggered prefetching* to dynamically convert the relevant data to RDF, using the transformations briefly described above. We will illustrate this by means of an example scenario where a user is interested in documents about ”AIDS” and does a number of subsequent queries.

The user enters the search string (”AIDS”) in the DOPE client. To disambiguate the search string (that is, to find the relevant thesaurus keyword concept), the client sends the following SeRQL query:

```
select ConceptName, Concept
from {Concept} dope:like {"AIDS"};
               rdfs:label {ConceptName}
```

The query is received by the mediator, which examines it. The use of the property `dope:like` invokes a trigger: the mediator starts extracting keywords that, according to the Collexis server, match the phrase ”AIDS”.

It is important to note that this *triggered prefetching* process is essentially the same as selecting a relevant sub query for the source to retrieve the information. Conceptually, the above query is translated to an equivalent query in the source repository’s own terminological model:

```
{Concept} clx:matchesSearchString {"AIDS"};
          clx:conceptName {ConceptName}
```

[7] `http://ontoWeb.aifb.uni-karlsruhe.de/Ontology`

However, the DOPE mediator takes an implementational shortcut: it immediately invokes an information extraction process through the metadata server's SOAP interface.

The Collexis server returns the keywords as an XML document, which is translated to an RDF model of the following form:

```
emtree:35079 clx:matchesSearchString "AIDS".
emtree:35079 rdf:type clx:Concept.
emtree:35079 clx:conceptName "Acquired Immune Deficiency Syndrome".
emtree:49320 clx:matchesSearchString"AIDS".
emtree:49320 rdf:type clx:Concept.
emtree:49320 clx: conceptName  "Visual Aids".
```

This RDF model is a physical model and uses terminology from the Collexis RDF schema (Figure 8.6). The next step transforms the physical model into a new RDF model in terms of the logical schema. It performs this translation using a SeRQL construct query that maps the source terminology back to the original terms of the user query:

```
construct {Concept} rdfs:label {Name};
                    dope:like {SearchString}
from {Concept} clx:conceptName {Name};
               clx:matchesSearchString {SearchString}
```

Applying this transformation query yields the following result:

```
emtree:35079 dope:like "AIDS".
emtree:35079 rdfs:label "Acquired Immune Deficiency Syndrome".
emtree:49320 dope:like "AIDS".
emtree:49320 rdfs:label "Visual Aids".
```

This data represents the same information, but now in terms of the logical model in which the original SeRQL query was formulated. The logical model is stored in the document repository, which now contains the information necessary to answer the DOPE clients query. The DOPE client receives the requested list of keywords and presents it to the user, who chooses a concept. The DOPE client then sends a new query to retrieve the documents related to the chosen keyword and the documents metadata:

```
select Document, URL, Title, ...
from {Document} dope:index {emtree:35079};
                dope:source {URL};
                ow:title {Title};
...
```

Again, the query engine decomposes this query, and the mediator forwards each subcomponent independently to each relevant source. Because this information hasnt been retrieved before, the mediator starts a new extraction process to retrieve it from the Collexis server and translates it in two steps into a logical model.

After retrieving the answer to the second query, the DOPE client needs, for each document, a list of related concepts. The third and final query retrieves these concepts:

```
select RelatedConcept, ConceptName, Doc
from {Doc} dope:index {emtree:35079, RelatedConcept},
     {RelatedConcept} rdfs:label {ConceptName}
```

Since the previous query already retrieved and stored the `dope:index` property relations for each document (they were *prefetched*), we can immediately evaluate the query against the logical model. The mediator forwards the call to the logical model directly instead of starting another extraction process from the Collexis server. The `rdfs:label` relations are available in the EMTREE repository, so the mediator forwards the subquery to that repository. The SeRQL query engine reintegrates the distributed results.

Discussion

The DOPE prototype is a simple implementation of distributed querying: only limited query decomposition takes place, since for many queries the complete query can be answered by a single source. The nature of the data distribution is such that this is possible. This simplistic nature of the data distribution has also made it possible for the DOPE system to employ a very simple indexing scheme for paths: since certain types of knowledge are always isolated in a single source, all the mediator has to do is examine queries for the presence of single properties, rather than complete path expressions. In a generalized setting, or in an extended version of DOPE (with additional data sources), this approach will have to be revised with a full dynamic indexing scheme, where the mediator examines each local source to index the knowledge they contain, and use that knowledge for determining triggers.

The fact that the prototype system has a dedicated user interface that uses a fixed set of queries has made it possible to introduce a prefetching and triggering strategy that anticipates the following queries by filling the logical model (which is, in effect, a cache for the Collexis server's data). The system can effectively anticipate precisely because it has knowledge of the order in which the user interface executes queries. In a more general setting, the prefetching and triggering strategy would have to be generalized or even abolished. In a series of tests involving end users of the system, the performance has been shown to be sufficient for interactive use. For details on this user test we refer the reader to [Stuckenschmidt et al., 2004a].

Nevertheless, the path expression-based triggering of sub queries in the mediator is very much in line with the general model of distributed querying through the Sesame framework that we have seen in section 8.3.

8.5 Related Work: A Generic Mediator SAIL

In [Adamku, 2004], Gergely Adamku introduces a prototype implementation of a generic distributed querying system implemented in the Sesame framework. We will briefly discuss his approach here.

In Adamku's system, the mediator is an implementation of Sesame's SAIL API, which can be configured through an XML file to connect to several local stores, either locally or remote (through Sesame's HTTP interface). This approach makes integrating new sources relatively easy: it only requires adapting the XML configuration file and restarting the mediator system.

The system uses a simple approach for query distribution, where each query is broken down into single triple patterns, and each pattern request is forward to local sources. Two strategies are implemented for this forwarding:

- *Brute-force*, or *flooding*: the mediator forwards each incoming request to every known local source, and integrates the results it gets back into a single result.
- *Predicate-Indexing*: in this approach, the mediator has built an index of predicate occurrences in each local source. Whenever a query is evaluated, the mediator examines the predicate index to determine which local sources have data that might satisfy the query, and it only forwards the request to these sources.

Several optimizations of these basic approaches are discussed in [Adamku, 2004], and an evaluation of the prototype on a few test datasets is given. Unsurprisingly, due to the basic nature of the indexing scheme and the fact that no query optimization techniques are implemented, the system performs relatively poorly when benchmarked against the performance of a single repository. As expected, the flooding strategy performs poorest, and the predicate indexing approach performs slightly better but worse than querying locally only.

Although the author concludes that more sophisticated indexing and forwarding techniques are required to improve performance, it should be pointed out that the approach has merit in itself: while performance is lower than the local store, *scalability* of the approach is potentially much higher. While this has not been tested in the evaluation in [Adamku, 2004], we suspect that on data sets of sufficiently large scale, the system will perform better relative to the local store, for the simple reason that the load can be shared accross different physical locations. The penalty for network traffic and result integration that is inherent in distributed querying may, in such a scenario, be offset by the fact that a single local repository can not scale beyond a certain size. We conclude that the Mediator prototype provides a basic implementation that could very easily be further extended within the context of the Sesame framework.

8.6 Future Work

In the presented case study, the knowledge on how to distribute queries was hardcoded into the mediator. The reason this was doable was that a limited number of distributed sources was present and the mappings were expected to be static for the lifetime of the project.

However, in many cases a more flexible approach will be required, allowing more declarative, dynamic mappings between global ontology and local information source. In [Vdovjak et al., 2003], an approach is sketched for providing such mappings. The authors introduce an *Integration Ontology* which can be used to define *articulations*, that is, mappings between global ontology and source schema. These articulations are provided as mappings between path expressions in the global and the local ontology. Current ongoing (and future) work focuses on implementing this strategy in the Sesame framework, taking the prototype mediator implementation of [Adamku, 2004], presented in the previous section, as a basis.

8.7 Conclusions

In the previous section, we have seen how distributed querying and information integration tasks can be augmented by using ontologies. We have shown how the Sesame framework can technically realize an infrastructure in which such tasks can be implemented, and have discussed a case study, the DOPE project, in which such a distributed system has been set up.

We conclude that a simple distributed system can easily be realized through the Sesame framework. The benefits of such an approach are higher scalability and being able to integrate various heterogeneous data sources without the need for physical data duplication. Future work will focus on further developing a generic mediator component in the framework, that can be configured through declarative source-to-global mappings.

Chapter 9

Conclusions

In this thesis we have investigated how to represent domain knowledge on the Web in a machine processable fashion, and storing, querying and reasoning with such information.

The following questions, stated in the introduction, have guided the discussion of proposed solutions in this thesis:

A: How do we represent machine processable information on the Web?

B: How do we access machine processable information on the Web?

C: How do we create tools that manipulate machine processable information on the Web?

In part I of this thesis, we have studied the first two of these questions, and in part II, we have investigated how to implement the languages that were introduced in part I. The contributions of this thesis that followed from these research questions have already been mentioned in the introduction, section 1.3:

1. Web-Based Knowledge Representation by Extending Existing Web Formalisms
2. A Query Language for RDF
3. Access APIs for manipulating and storing RDF
4. Inferencing Strategies for RDF

We will discuss each of these contributions in more detail here.

9.1 Discussion of Contributions

Web-Based Knowledge Representation by Extending Existing Web Formalisms

In chapter 2, we have addressed research question A. We discussed several web formalisms, concluding with RDF and RDF Schema as the most logical choice for a basis

of the Semantic Web. It was then argued that for full-fledged ontological modeling, RDF Schema is not enough and additional modeling primitives are required. We then proceeded to show how an expressive ontology language, OIL, can be layered on top of RDF Schema in such a way that both forward and backward compatibility are maximized, and we have illustrated how this compatibility maximization is better preserved in OIL than in its successor languages DAML+OIL and OWL.

A Query Language for RDF

In chapter 3, we have addressed research question B. We have outlined general requirements for an RDF query language and presented test findings on the extent to which several existing query language proposals conform to these requirements. In chapter 4, we proceeded to propose the SeRQL query language. We argued that its benefits include ease of use, formal specification and high expressiveness, and have shown how SeRQL compares with existing proposals. We also recognized that future work remains to be done on the SeRQL specification, and have outlined several specific areas for such future work.

Access APIs for manipulating and storing RDF

In chapter 5 we have addressed research question C. We have described the Sesame framework, which includes APIs, query language implementations and storage mechanisms for RDF. We have shown the benefits of the layered architecture of Sesame, and we have presented the notion of a generic query object model and the benefits it offers in terms of flexibility and optimization options.

Sesame offers a very flexible, stable and scalable platform for Semantic Web tool development in Java. It is already widely deployed in numerous projects and is one of the most-used RDF tools available.

Inferencing Strategies for RDF

In chapter 6, we have further addressed research question C. We have discussed a pruning iterative forward chaining entailment algorithm for RDF(S), and have shown its performance and scalability by means of benchmark tests. We have identified its shortcomings and have proceeded to analyze the tradeoff that exists between storage space and response time in such algorithms. Finally, we proposed an alternative algorithm that uses part forward chaining, part query rewriting to support RDFS reasoning. Concluding, we can say that the choice of reasoning strategy is dependent on the application domain: in a domain where the emphasis lies on querying, the data graph is relatively stable, and storage space is not a major concern, the full forward chaining approach will perform best. Whenever the data becomes more volatile or storage issues come into play, a hybrid approach like the schema-only reasoning algorithm presented will be the better option.

In chapter 7, we further addressed one of the side effects of using forward chaining inferencing: the notion of disbelief propagation, or more generally, truth maintenance. We discussed three different strategies for dealing with truth maintenance and have shown

benchmark tests for their performance. The conclusion is again that different scenarios require different strategies. The dependency-based algorithm works best in scenarios with medium-size hierarchies. For larger hierarchies, the backward chaining truth maintenance algorithm may prove to be preferable.

Chapter 8 illustrates the contributions made by this thesis in a real world setting. The notion of using the Sesame framework is explored in a large use case in a production environment, namely the DOPE project.

9.2 Future Work

The Semantic Web research field is a young and fast-moving one, and consequently there are numerous directions for future research to take. We can identify a number of future research topics that will be crucial in the further development of Semantic Web tools.

9.2.1 Context Support

The notion of context in terms of RDF graphs can be very useful to control and manipulate large, heterogeneous sets of RDF data. Although not part of the specifications of RDF, many RDF systems provide some form of provenance/context support, where the source of a statement (i.e., the URI of the document from which the statement originally comes) is stored as a facet of each statement.

Although RDF's reification mechanism is conceptually able to encode such information, in practice this may not be desirable: reification adds four additional RDF statements for each reified triple. To use this mechanism on every triple would therefore lead to an unacceptable increase in the size of the repository.

A non-semantical explicit form of *context support*, where an additional property of each statement is automatically and efficiently stored in the backend (e.g. as an extra column in a database table) and exposed to the world *as if* it were a form of reification, is a desirable option to have in an RDF repository: it will enable the grouping of sets of statements according to source, timestamp, version or whatever other grouping characteristic the user finds convenient, conceptually staying within the RDF specifications by making this information queryable as if it were reification, but in the backend making sure that a more compact specialized form of storage is used.

Future work on context support should nvestigate two directions:

- extending the Sesame Framework with implementation support for a notion of context.
- extending the RDF Model specification itself with a standardized abstract notion of context.

9.2.2 OWL Reasoning

In this thesis we have presented algorithms for reasoning with RDF and RDF Schema, and a framework in which these algorithms are implemented. However, in the first part of

the thesis we have outlined the need for more expressive knowledge representation primitives than RDF Schema can offer. These primitives are available in the Web Ontology Language OWL, the successor of the OIL language.

As described in chapter 2, OWL, like OIL, is an extension of RDF Schema. Nevertheless, reasoning with OWL ontologies is considerably more complex than for RDF Schema: it requires not only entailment reasoning but also constraint satisfaction checking, and moreover its expressivity goes beyond Horn clauses and is therefore not possible to capture through a simple rule-based approach.

We can distinguish two major approaches to OWL reasoning, each of which we will briefly discuss here.

Using Description Logic Reasoners

OWL DL is a subset of OWL that corresponds to an expressive Description Logic (more precisely, the $\mathcal{SHOIQ}(\mathcal{D})$ DL [Horrocks and Satler, 2001]). A Description Logic is a decidable fragment of FOL, and efficient algorithms are known and implemented for it - examples are the Racer [Haarslev and Möller, 2001] and FaCT [Horrocks, 1999] theorem provers.

The combination of such a theorem prover with a framework such as Sesame, however, is not a trivial task. DL reasoners do not expect content to be delivered in a triple format, which means conversions will have to take place. The interaction pattern between store and reasoning system will have to be analysed to produce an integration of the tools that performs in real world settings.

Using Rule-Based Reasoners

In [Grosof et al., 2003] a subset of OWL DL is identified that corresponds to the intersection between Description Logics and Horn Logic. It is argued that this subset, dubbed OWL DLP (for Description Logic Programs), is sufficiently restricted to be efficiently implemented by a (Horn)rule-based system.

In general, rule-based reasoners can go a long way in supporting subsets of OWL that, although strictly speaking not sound and complete, can be practically useful in many applications in which only subsets of the OWL vocabulary are required. Identifying such practical subsets, and use cases in which they are applicable, is therefore another direction for future work on OWL support.

9.2.3 Rules, Views and Transformations

In chapter 4 we have shown how SeRQL is a simple graph transformation language. Future work could include research into how SeRQL can be further extended to be an ”XSLT for RDF”, that is, a full graph transformation language that can transform any incoming graph to any outcoming graph. Possible extensions to think of include a more mature mechanism for introducing new object identifiers (URIs and blank nodes) in the transformation, and features such as recursion and matching of arbitrary-length paths.

In chapter 8, it was pointed out that strictly speaking SeRQL is not a view definition language, since it does not have a mechanism for naming queries. Investigating how such view definitions could be realized in terms of SeRQL is non-trivial: not only does it require a conceptual extension of the language, it also requires investigation into realization and materialization of views in the context of the Sesame framework.

Finally, work on rule languages for RDF and OWL, such as SWRL [Horrocks et al., 2003] bears close resemblances to work on SeRQL and other query languages (not surprisingly, since a query is essentially a simple type of rule). Future work on unifying these approach in a language that is applicable for both querying and rule formulation would be an interesting step towards further integration of the plethora of proposals for languages and tools, and help solidify the basis of the Semantic Web further.

Bibliography

[Abiteboul et al., 1999] Abiteboul, S., Buneman, P., and Suciu, D. (1999). *Data on the Web: From Relations to Semistructural Data and XML*. Morgan Kaufman.

[Adamku, 2004] Adamku, G. (2004). Implementation and Evaluation of Distributed RDF Querying. Msc thesis, Vrije Universiteit, Amsterdam.

[Adler et al., 2000] Adler, S., Adler, S., Berglund, A., Caruso, J., Deach, S., Grosso, P., Gutentag, E., Milowski, A., Parnell, S., Richman, J., , and Zilles, S. (2000). Extensible Stylesheet Language (XSL). Working Draft, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/xsl/`.

[Alexaki et al., 2000] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K. (2000). The RDFSuite: Managing Voluminous RDF Description Bases. Technical report, Institute of Computer Science, FORTH, Heraklion, Greece. See `http://www.ics.forth.gr/proj/isst/RDF/RSSDB/rdfsuite.pdf`.

[Bechhofer, 2003] Bechhofer, S. (2003). The DIG Interface. See `http://dig.sourceforge.net`.

[Bechhofer et al., 1999] Bechhofer, S., Horrocks, I., Patel-Schneider, P. F., and Tessaris, S. (1999). A proposal for a description logic interface. In *Proc. of DL'99*, pages 33–36.

[Beckett, 2001] Beckett, D. (2001). The Design and Implementation of the Redland RDF Application Framework. In *Proceedings of Semantic Web Workshop of the 10th International World Wide Web Conference*, Hong-Kong, China.

[Beckett, 2004] Beckett, D. (2004). RDF/XML Syntax Specification (Revised). Recommendation, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/rdf-syntax-grammar/`.

[Berners-Lee, 1998a] Berners-Lee, T. (1998a). Notation 3. See `http://www.w3.org/DesignIssues/Notation3.html`.

[Berners-Lee, 1998b] Berners-Lee, T. (1998b). Semantic Web Road map. Internal note, World Wide Web Consortium (W3C). See `http://www.w3.org/DesignIssues/Semantic.html`.

[Berners-Lee, 2000] Berners-Lee, T. (2000). CWM - closed world machine. See http://www.w3.org/2000/10/swap/doc/cwm.html.

[Biron and Malhotra, 2001] Biron, P. V. and Malhotra, A. (2001). XML Schema Part 2: Datatypes. Recommendation, World Wide Web Consortium. See http://www.w3.org/TR/xmlschema-2/.

[Boag et al., 2005] Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., and Simeon, J. (2005). XQuery: An XML Query Language. Working draft, World Wide Web Consortium. See http://www.w3.org/TR/xquery/.

[Borgida et al., 1989] Borgida, A., Brachman, R., McGuinness, D., and Resnick, L. (1989). CLASSIC: A Structural Data Model for Objects. In *SIGMOD Conference*.

[Bozsak et al., 2002] Bozsak, E., Ehrig, M., Handschuh, S., Hotho, A., and Maedche, A. (2002). KAON - Towards a Large Scale Semantic Web. In *EC-Web 2002*.

[Brachman and Schmolze, 1985] Brachman, R. J. and Schmolze, J. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216.

[Bray et al., 1998] Bray, T., Paoli, J., and Sperberg-McQueen, C. (1998). Extensible Markup Language (XML) 1.0. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/1998/REC-xml-19980210.

[Brickley and Guha, 2000] Brickley, D. and Guha, R. (2000). Resource Description Framework (RDF) Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium. See http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

[Brickley and Guha, 2004] Brickley, D. and Guha, R. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[Broekstra et al., 2003] Broekstra, J., Ehrig, M., Haase, P., v. Harmelen, F., Kampman, A., Sabou, M., Siebes, R., Staab, S., Stuckenschmidt, H., and Tempich, C. (2003). A Metadata Model for Semantics-Based Peer-to-Peer Systems. In *1st Workshop on Semantics in Peer-to-Peer and Grid Computing at the Twelfth International World Wide Web Conference*, Budapest, Hungary. See http://swap.semanticweb.org/.

[Broekstra et al., 2000] Broekstra, J., Fluit, C., and van Harmelen, F. (2000). The State of the Art on Representation and Query Languages for Semistructured Data. On-To-Knowledge (IST-1999-10132) Deliverable 8, Aidministrator Nederland b.v. See http://www.ontoknowledge.org/.

[Broekstra and Kampman., 2003] Broekstra, J. and Kampman., A. (2003). Inferencing and Truth Maintenance in RDF Schema: Exploring a Naive Practical Approach. In *Workshop on Practical and Scalable Semantic Systems (PSSS) at the Second International Semantic Web Conference (ISWC)*, Sanibel Island, Florida,.

[Broekstra and Kampman, 2003] Broekstra, J. and Kampman, A. (2003). The SeRQL Query Language. Technical report, Aduna. See `http://www.openrdf.org/doc/SeRQLmanual.html`.

[Broekstra and Kampman, 2004] Broekstra, J. and Kampman, A. (2004). SeRQL: An RDF Query and Transformation Language. To be published. Draft available at `http://www.cs.vu.nl/~jbroeks/papers/SeRQL.pdf`.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Horrocks, I. and Hendler, J., editors, *Proceedings of the first International Semantic Web Conference (ISWC 2002)*, number 2342 in Lecture Notes in Computer Science, pages 54–68, Sardinia, Italy. Springer Verlag, Heidelberg Germany. See also `http://www.openrdf.org/`.

[Broekstra et al., 2001] Broekstra, J., Klein, M., Decker, S., Fensel, D., van Harmelen, F., and Horrocks, I. (2001). Enabling Knowledge Representation on the Web by Extending RDF Schema. In *Proceedings of the 10th International World Wide Web Conference*, Hong-Kong, China.

[Carrol and McBride, 2001] Carrol, J. and McBride, B. (2001). The Jena Semantic Web Toolkit. Public api, HP-Labs, Bristol. See `http://www.hpl.hp.com/semweb/jena-top.html`.

[Christophides et al., 2003] Christophides, V., Plexousakis, D., Scholl, M., and Tourtounis, S. (2003). On Labeling Schemes for the Semantic Web. In *Proc. of the 12th International World Wide Web Conference (WWW'03)*, pages 544–555, Budapest, Hungary.

[de Kleer, 1986] de Kleer, J. (1986). An Assumption-based TMS. *Artificial Intelligence*, 28(2).

[Dean and Schreijber, 2004] Dean, M. and Schreijber, G. (2004). OWL Web Ontology Language Reference. Recommendation, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/owl-ref/`.

[Decker et al., 1999] Decker, S., Erdmann, M., Fensel, D., and Studer, R. (1999). Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. In Meersman, R., editor, *Database Semantics, Semantic Issues in Multimedia Systems*, pages 351–369. Kluwer Academic Publisher, Boston.

[Deutsch et al., 1998] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1998). XML-QL: A Query Language for XML. Position paper QL'98 Workshop, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[Doyle, 1979] Doyle, J. (1979). A Truth Maintenance System. *Artificial Intelligence*, 12.

[Fallside and Walsmley, 2004] Fallside, D. and Walsmley, P. (2004). XML Schema Part 0: Primer. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/xmlschema-0/.

[Fensel, 2000] Fensel, D. (2000). *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, Berlin.

[Fensel et al., 1998] Fensel, D., Decker, S., Erdmann, M., and Studer, R. (1998). Ontobroker: The Very High Idea. In *Proceedings of the 11th International Flairs Conference (FLAIRS-98)*, Sanibel Island, Florida.

[Fensel et al., 2000a] Fensel, D., Horrocks, I., van Harmelen, F., Decker, S., Erdmann, M., and Klein, M. (2000a). OIL in a nutshell. In Dieng, R. and Corby, O., editors, *Knowledge Engineering and Knowledge Management; Methods, Models and Tools, Proceedings of the 12th International Conference EKAW 2000*, number LNCS 1937 in Lecture Notes in Artificial Intelligence, pages 1–16, Juan-les-Pins, France. Springer-Verlag.

[Fensel and Perez, 2002] Fensel, D. and Perez, A. (2002). A survey on ontology tools. Technical Report OntoWeb Deliverable 1.3, OntoWeb consortium. See http://www.ontoweb.org/download/deliverables/D13_v1-0.zip.

[Fensel et al., 2000b] Fensel, D., van Harmelen, F., Klein, M., Akkermans, H., Broekstra, J., Fluit, C., van der Meer, J., Schnurr, H.-P., Studer, R., Hughes, J., Krohn, U., Davies, J., Engels, R., Bremdal, B., Ygge, F., Lau, T., Novotny, B., Reimer, U., and Horrocks, I. (2000b). On-To-Knowledge: Ontology-based Tools for Knowledge Management. In *Proceedings of the eBusiness and eWork 2000 (EMMSEC 2000) Conference*, Madrid, Spain.

[Forgy, 1982] Forgy, C. (1982). Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37.

[Garcia-Milina et al., 1997] Garcia-Milina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., and Ullman, J. (1997). Ths TSIMMIS project: Integation of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132.

[Grant and Beckett, 2003] Grant, J. and Beckett, D. (2003). RDF Test Cases. Proposed recommendation, World Wide Web Consortium. See http://www.w3.org/TR/rdf-testcases/.

[Grosof et al., 2003] Grosof, B., Horrocks, I., Volz, R., and Decker, S. (2003). Description logic programs: Combining logic programs with description logics. In *Proceedings of the World Wide Web Conference (WWW2003)*, Budapest, Hungary.

[Gruber, 1993] Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2).

[Guo et al., 2004] Guo, Y., Pan, Z., and Heflin, J. (2004). An Evaluation of Knowledge Base Systems for Large OWL Datasets. In Plexousakis, D., McIlraith, S., and van Harmelen, F., editors, *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science. Springer.

[Gutierrez et al., 2004] Gutierrez, C., Hurtado, C. A., and Mendelzon, A. O. (2004). Foundations of semantic web databases. In *Proceedings of the Twenty-third Symposium on Principles of Database Systems (PODS), June 14-16, 2004, Paris, France*, pages 95–106.

[Haarslev and Möller, 2001] Haarslev, V. and Möller, R. (2001). RACER System Description. In Goré, R., Leitsch, A., and Nipkow, T., editors, *Automated Reasoning: First International Joint Conference (IJCAR) 2001*, volume 2083 of *Lecture Notes in Computer Science*, page 701, Siena, Italy. Springer-Verlag.

[Haase et al., 2004] Haase, P., Broekstra, J., Eberhart, A., and Volz, R. (2004). A Comparison of RDF Query Languages. In McIlraith, S., Plexousakis, D., and van Harmelen, F., editors, *The Semantic Web - ISWC 2004. Proceedings of the Third International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, Hiroshima, Japan. Springer-Verlag.

[Halevy, 2001] Halevy, A. (2001). Answering queries using views – a survey. *The VLDB Journal*, 10(4):270–294.

[Hayes and Gutierrez, 2004] Hayes, J. and Gutierrez, C. (2004). Bipartite Graphs as Intermediate Model for RDF. In McIlraith, S., Plexousakis, D., and van Harmelen, F., editors, *The Semantic Web - ISWC 2004. Proceedings of the Third International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 47–61, Hiroshima, Japan. Springer-Verlag.

[Hayes, 2004] Hayes, P. (2004). RDF Semantics. Recommendation, World Wide Web Consortium. See `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

[Heflin and Hendler, 2000] Heflin, J. and Hendler, J. (2000). Dynamic Ontologies on the Web. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449. AAAI/MIT Press, Menlo Park, CA.

[Hendler, 2001] Hendler, J. (2001). Agents and the semantic web. *IEEE Intelligent Systems*, 16(2).

[Horrocks, 1999] Horrocks, I. (1999). FaCT and iFaCT. In *Proceedings of the Description Logics Workshop, DL'99*.

[Horrocks et al., 2000] Horrocks, I., Fensel, D., Broekstra, J., Decker, S., Erdmann, M., Goble, C., van Harmelen, F., Klein, M., Staab, S., Studer, R., and Motta, E. (2000). OIL: The Ontology Inference Layer. Technical Report IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences. See `http://www.ontoknowledge.org/oil/`.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., and andM. Dean, B. G. (2003). SWRL: A semantic web rule language combining OWL and RuleML. see `http://www.daml.org/2003/11/swrl/`.

[Horrocks and Satler, 2001] Horrocks, I. and Satler, U. (2001). Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of IJCAI 2001*, pages 199–204.

[Horrocks et al., 2001] Horrocks, I., van Harmelen, F., Patel-Schneider, P., Berners-Lee, T., Brickley, D., Connoly, D., Dean, M., Decker, S., Fensel, D., Hayes, P., Heflin, J., Hendler, J., Lassila, O., McGuinness, D., and Stein, L. A. (2001). DAML+OIL. `http://www.daml.org/2001/03/daml+oil-index.html`.

[Hoschka, 1998] Hoschka, P. (1998). Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. Recommendation, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/REC-smil`.

[ISO8879:1986, 1986] ISO8879:1986 (1986). Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML). Standard No. ISO 8879:1986, International Organization for Standardization.

[Karvounarakis et al., 2002] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Schol, M. (2002). RQL: A Declarative Query Language for RDF. In *Proceedings of the Eleventh International World Wide Web Conference (WWW'02)*, Honolulu, Hawaii, USA.

[Kifer et al., 1995] Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42.

[Kiryakov et al., 2002] Kiryakov, A., Simov, K. I., and Ognyanov, D. (2002). Ontology Middleware: Analysis and Design. On-To-Knowledge (IST-1999-10132) Deliverable 38, OntoText. See `http://www.ontotext.com/publications/index.html#KiryakovEtAl2002`.

[Klein et al., 2000] Klein, M., Fensel, D., van Harmelen, F., and Horrocks, I. (2000). The Relation between Ontologies and Schema-Languages: Translating OIL-Specifications in XML-Schema. In Benjamins, V. R., Gomez-Perez, A., and Guarino, N., editors, *Proceedings of the Workshop on Applications of Ontologies and Problem-solving Methods, 14th European Conference on Artificial Intelligence (ECAI 2000)*, Berlin, Germany.

[Klyne and Carroll, 2004] Klyne, G. and Carroll, J. (2004). Resource Description Framework (RDF): Concepts and Abstract Data Model. Recommendation, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/rdf-concepts/`.

[Lassila, 2002] Lassila, O. (2002). Taking the RDF Model Theory Out for a Spin. In Horrocks, I. and Hendler, J., editors, *Proceedings of the First International Semantic Web Conference, ISWC 2002, Sardinia, Italy*, number 2342 in Lecture Notes in Computer Science, pages 307–317. Springer-Verlag, Heidelberg, Germany.

[Lassila and Swick, 1999] Lassila, O. and Swick, R. R. (1999). Resource Description Framework (RDF): Model and Syntax Specification. Recommendation, World Wide Web Consortium (W3C). See `http://www.w3.org/TR/REC-rdf-syntax/`.

[Levy, 1999] Levy, A. (1999). Combining artificial intelligence and databases for data integration. In Wooldridge, M. and Veloso, M., editors, *Artificial Intelligence Today: Recent Trends and Developments*, number 1600 in Lecture Notes in Computer Science, pages 249–268. Springer-Verlag, Heidelberg, Germany.

[Levy et al., 1996] Levy, A., Rajaraman, A., and Ordille, J. (1996). Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases, VLDB-96*, pages 251–262, Bombay, India.

[Luke et al., 1996] Luke, S., Spector, L., and Rager, D. (1996). Ontology-Based Knowledge Discovery on the World-Wide Web. In Franz, A. and Kitano, H., editors, *Working Notes of the Workshop on Internet-Based Information Systems at the 13th National Conference on Artificial Intelligence (AAAI96)*, pages 96–102. AAAI Press.

[Maganaraki et al., 2002] Maganaraki, A., Karvounarakis, G., Christophides, V., Plexousakis, D., and Anh, T. (2002). Ontology storage and querying. Technical Report 308, Foundation for Research and Technology Hellas, Institute of Computer Science, Information Systems Laboratory.

[Malhotra et al., 2003] Malhotra, A., Melton, J., and Walsh, N. (2003). Xquery 1.0 and xpath 2.0 functions and operators, w3c working draft 12 november 2003. See `http://www.w3.org/TR/xpath-functions/`.

[Martins, 1990] Martins, J. (1990). The Truth, the Whole Truth and Nothing But the Truth. *AI Magazine: Special Issue*, 11(7).

[Miller, 2001] Miller, L. (2001). RDF Squish query language and Java implementation. Public draft, Institute for Learning and Research Technology. See `http://ilrt.org/discovery/2001/02/squish/`.

[Mylopoulos et al., 1990] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. (1990). Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4).

[Nejdl et al., 2000] Nejdl, W., Wolpers, M., and Capella, C. (2000). The RDF Schema Revisited. In *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik, Modellierung 2000, St. Goar*. Foelbach Verlag, Koblenz.

[Reggiori and Seaborne, 2002] Reggiori, A. and Seaborne, A. (2002). Query and rule languages use cases and examples. See `http://rdfstore.sourceforge.net/2002/06/24/rdf-query/query-use-cases.html`.

[Robie et al., 1998] Robie, J., Lapp, J., and Schach, D. (1998). XML Query Language (XQL). Position paper QL'98 Workshop, World Wide Web Consortium (W3C). See `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[Roo, 2002] Roo, J. D. (2002). Euler proof mechanism. See http://www.agfa.com/w3c/euler/.

[Rothnie et al., 1980] Rothnie, J. B., Bernstein, P. A., Fox, S., Goodman, N., Hammer, M., Landers, T. A., Reeve, C., Shipman, D. W., and Wong, E. (1980). Introduction to a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):1–17.

[Seaborne, 2004] Seaborne, A. (2004). RDQL - A Query Language for RDF. W3c member submission, Hewlett Packard. See http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[Sintek and Decker, 2001] Sintek, M. and Decker, S. (2001). TRIPLE - an RDF query, inference and transformation language. In *Deductive Databases and Knowledge Management (DDLP)*.

[Staab et al., 2000] Staab, S., Erdmann, M., Mädche, A., and Decker, S. (2000). An Extensible Approach for Modeling Ontologies in RDF(S). In *First Workshop on the Semantic Web at the Fourth European Conference on Digital Libraries, Lisbon, Portugal*.

[Staab and Mädche, 2000] Staab, S. and Mädche, A. (2000). Axioms are Objects, too - Ontology Engineering beyond the Modeling of Concepts and Relations. In Benjamins, V., Gomez-Perez, A., and Guarino, N., editors, *Proceedings of the Workshop on Applications of Ontologies and Problem-solving Methods, 14th European Conference on Artificial Intelligence ECAI 2000, Berlin, Germany*.

[Stein et al., 2000] Stein, L. A., Connolly, D., and McGuinness, D. (2000). DAML-ONT Initial Release. http://www.daml.org/2000/10/daml-ont.html.

[Stuckenschmidt, 2000] Stuckenschmidt, H. (2000). Using OIL for Intelligent Information Integration. In Benjamins, V., Gomez-Perez, A., and Guarino, N., editors, *Proceedings of the Workshop on Applications of Ontologies and Problem-solving Methods, 14th European Conference on Artificial Intelligence ECAI 2000, Berlin, Germany*.

[Stuckenschmidt, 2003] Stuckenschmidt, H. (2003). *Ontology-Based Information Sharing in Weakly Structured Environments*. PhD thesis, Vrije Universiteit, Amsterdam.

[Stuckenschmidt and Broekstra, 2005] Stuckenschmidt, H. and Broekstra, J. (2005). Time-space Tradeoffs in RDF Schema Reasoning. To be published.

[Stuckenschmidt et al., 2004a] Stuckenschmidt, H., de Waard, A., Bhogal, R., Fluit, C., Kampman, A., van Buel, J., van Mulligen, E., Broekstra, J., Crowlesmith, I., van Harmelen, F., and Scerri, T. (2004a). Exploring Large Document Repositories with RDF Technology - The DOPE Project. *IEEE Intelligent Systems - Special Issue on the Semantic Web Challenge*.

[Stuckenschmidt et al., 2004b] Stuckenschmidt, H., Vdovjak, R., Broekstra, J., and Houben, G.-J. (2004b). Data Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of the International World Wide Web Conference WWW'04*, New York, USA.

[ter Horst, 2004] ter Horst, H. (2004). Extending the RDFS Entailment Lemma. In Plexousakis, D., McIlraith, S., and van Harmelen, F., editors, *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science. Springer.

[Thompson et al., 2001] Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. (2001). XML Schema Part 1: Structures. Recommendation, World Wide Web Consortium. See `http://www.w3.org/TR/xmlschema-1/`.

[Ullman, 1997] Ullman, J. (1997). Information Integration Using Logical Views. In *Proceedings of the 6th International Conference on Database Theory*, pages 19–40.

[Vdovjak et al., 2003] Vdovjak, R., Barna, P., and Houben, G. (2003). Designing a Federated Multimedia Information System on the Semantic Web. In Eder, J. and Missikoff, M., editors, *Advanced Information System Engineering, 15th International Conference, CAiSE 2003*, volume 2681 of *Lecture Notes in Computer Science*, Klagenfurt/Velden, Austria. Springer-Verlag.

[Wache et al., 2001] Wache, H., Vögele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hübner, S. (2001). Ontology-Based Integration of Information – A Survey of Existing Approaches. In *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing*, Seattle, USA.

[Wielemaker et al., 2003] Wielemaker, J., Schreiber, G., and Wielinga, B. (2003). Prolog-Based Infrastructure for RDF: Scalability and Performance. In Fensel, D., Sycara, K., and Mylopoulos, J., editors, *The Semantic Web - ISWC 2003, Second International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 644 – 658.

[Wood et al., 2005] Wood, D., Gearon, P., and Adams, T. (2005). Kowari: A Platform for Semantic Web Storage and Analysis. In *Proceedings of the 14th World Wide Web Conference (WWW 2005)*, Chiba, Japan.

Samenvatting

Het *Semantic Web* is een uitbreiding van het huidige World Wide Web, waarbij informatie op een zodanige manier wordt verrijkt en gerepresenteerd dat computers zelfstandig die informatie kunnen verwerken en interpreteren, en hiermee kunnen redeneren. De voordelen van deze verbeterde manier van het delen van kennis zijn legio: computers zullen gebruikers doeltreffender kunnen assisteren bij het vinden van relevante informatie en zullen semi-zelfstandig informatie uit verschillende bronnen kunnen combineren en daaruit conclusies trekken.

Om het Semantic Web te kunnen realiseren is echter infrastructuur nodig:

- formele talen voor het modelleren van kennis;
- formele talen voor het benaderen van de gemodelleerde kennis;
- ontwikkelraamwerken die het voor software-ontwikkelaar mogelijk maken om kennis, opgeslagen en benaderbaar via deze talen, te integreren in applicaties.

Ontologieeën vormen hierbij een cruciale rol. Een ontologie is een formele specificatie van een gedeelde conceptualisatie, dat wil zeggen, een manier om kennis zodanig te modelleren en representeren dat de betekenis eenduidig vastligt, en dat er overeenstemming tussen betrokkenen over die vastlegging bestaat.

In dit proefschrift worden nieuwe formele talen en een ontwikkelraamwerk voor deze doeleinden gepresenteerd. Het eerste deel behandelt modelleertalen voor kennis en hun relatie met het Web. We beschrijven de basisprincipes van het Resource Description Framework, RDF, en de primitieve ontologietaal RDF Schema. Vervolgens behandelen we hoe door middel van een uitbreiding van RDF Schema een expressievere modelleertaal, zoals OIL, de bestaande webstandaarden als fundering kan gebruiken. De nadruk bij deze extensie ligt op het maximaliseren van de compatibiliteit tussen de lagen: een applicatie die slechts de semantiek van RDF/RDF Schema begrijpt kan een incomplete maar wel consistente interpretatie van een OIL ontologie doen. Omgekeerd geldt dat elke RDF Schema ontologie een correcte OIL ontologie is.

Als kennis is geformaliseerd door middel van een dergelijke modelleertaal zijn toegangsmechanismen zoals querytalen nodig om de kennis te kunnen toepassen in applicaties. In hoofdstuk 3 worden verscheidene querytalen onder de loep genomen. We presenteren algemene eisen waaraan een querytaal moet voldoen en een aantal kenmerken die een querytaal karakteriseren. Vervolgens worden door middel van een aantal

vragen op een voorbeeldmodel de capaciteiten van elke querytaal behandeld. In hoofdstuk 4 wordt een nieuwe querytaal, genaamd SeRQL, gepresenteerd. Het ontwerp van SeRQL is gebaseerd op ervaringen met implementatie en gebruik van andere querytalen. Het doel van de taal is het verenigen van de beste kenmerken van deze talen in één nieuwe taal. Daarnaast biedt SeRQL oplossingen voor praktische problemen die gerezen waren bij deze andere querytalen. Het ontwerp, de syntax en de formele interpretatie van SeRQL worden uitgebreid behandeld.

Het tweede deel van het proefschrift behandelt API's en raamwerken voor het opslaan, queryen en redeneren met RDF. In hoofdstuk 5 wordt Sesame gepresenteerd. Sesame is een ontwikkelraamwerk dat het mogelijk maakt grote hoeveelheden RDF efficiënt op te slaan en opnieuw te benaderen. We behandelen de gelaagde architectuur van Sesame in detail, met nadruk op de toegangsmechanismen van Sesame, in het bijzonder de access API's en de verschillende query engines (voor SeRQL, RQL en RDQL). Ook de opslagmechanismen van Sesame worden behandeld; via een simpel experiment worden de verschillen in capaciteit en prestatie geïllustreerd.

In hoofdstuk 6 gaan we dieper in op het redeneeraspect van RDF. RDF kent een simpele modeltheoretische semantiek die correspondeert met een verzameling Hornregels die beschrijven welke nieuwe statements uit bestaande statements kunnen worden afgeleid. We behandelen kort de eigenschappen van deze verzameling regels en presenteren een iteratief forward-chaining algoritme dat efficiënt deze regels kan toepassen. We demonstreren, via een implementatie van dit algoritme in het Sesame framework, hoe het presteert op verscheidene datasets.

In het vervolg van het hoofdstuk analyseren we knelpunten in deze aanpak. Een van de voornaamste knelpunten is de hoeveelheid data die door het algoritme wordt opgeslagen. We presenteren een alternatief algoritme dat een hybride is tussen forward-chaining redeneren en query herschrijven: het idee is dat het redeneeralgoritme incompleet is, en als zodanig minder data opslaat. Tijdens het evalueren van een query kan de ontbrekende data worden aangevuld door de query intern te herschrijven. De nadruk ligt op het aantonen van de correctheid van de oplossing, en enkele tests op verschillende datasets tonen de potentiële winst in termen van hoeveelheden opgeslagen statements aan.

Hoofdstuk 7 gaat nader in op een aspect van forward chaining redeneren: het bewaren van consistentie van de opgeslagen kennis. We presenteren een truth-maintenance-algoritme dat door middel van het offline genereren van de verzameling van afhankelijkheden tussen statements consistentie van de dataset bewaakt. In verscheidene tests worden voordelen en nadelen van de aanpak geïllustreerd. Uit deze tests blijkt dat een groot knelpunt het offline genereren van de verzameling afhankelijkheden is. De nadruk wordt gelegd op het feit dat voor verschillende taken, zoals versioning en security, deze afhankelijkheden noodzakelijk aanwezig is. Echter, voor situaties waarin deze taken geen rol spelen, wordt een alternatief algoritme gepresenteerd dat runtime via een backward-chaining strategie afhankelijkheden bepaalt. We tonen de correctheid van de aanpak aan en gaan kort in op de complexiteit van het algoritme en de verwachte prestatieverbetering.

Tenslotte wordt in hoofdstuk 8 ingegaan op het queryen van gedistribueerde data. We presenteren kort de algemene ideeën hierachter en illustreren hoe een gedistribueerd systeem kan worden gerealiseerd met behulp van Sesame. Vervolgens wordt een casus, het DOPE project, behandeld. Hier wordt met een praktijkvoorbeeld de aanpak

geïllustreerd. De nadruk hierbij ligt op het integreren van heterogene databronnen onder een geünificeerde ontologie en interface.

Samenvattend hebben we in dit proefschrift onderzocht hoe we formele kennis kunnen representeren op het Web, hoe we die kennis vervolgens toegankelijk kunnen maken, en hoe we systemen moeten creëren die met deze kennis om kunnen gaan. We hebben aangetoond dat een uitbreiding van RDF formele kennisrepresentatie op het Web mogelijk maakt, dat querytalen als SeRQL een geschikte manier zijn om deze kennis toegankelijk te maken, en we hebben geïllustreerd hoe het Sesame framework een platform biedt voor systemen die met kennis in RDF, benaderbaar via SeRQL, willen kunnen werken.

SIKS Dissertation Series

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) *Prototyping of CMS Storage Management*

2000-3 Carolien M.T. Metselaar (UvA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectie

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupé (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VU)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel (KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (UvA)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UvA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek (UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications